Algorithmique Avancée

TD 1-2 : Structures de données avancées

Exemplaire enseignant (corrigé modifié en sept 2018)

Table des matières

1 Rappels : notations asymptotiques

Exercice 1.1 ***: Notations de Landau***

Question 1.1.1 Rappeler la formule du binôme de Newton. Montrer les égalités suivantes [1] :

$$\begin{aligned} 2^r &= \sum_{i=0}^{r} C_r^i \\ 2^r - 1 &= 2^{r-1} + \ldots + 2^2 + 2 + 1 \end{aligned}$$

Question 1.1.2 Rappeler les définitions des notations de Landau, O, Ω et Θ, pour les fonctions à valeur dans $\mathbb{R}_+^*$.

Question 1.1.3 Montrer que si $f \in O(g)$, alors $O(f) \subset O(g)$, et $O(max(f,g)) = O(g)$, où $max(f,g)$ vérifie : $\forall x \in \mathbb{R},\ max(f,g)(x) = max(f(x), g(x))$.

Question 1.1.4 Montrer que $O(f+g) = O(max(f,g))$.

Question 1.1.5 Montrer que :

$$\sum_{p=1}^{n} p^k = \Theta(n^{k+1}).$$

Solution

1. C_r^i, aussi noté $\binom{r}{i}$, est un coefficient binomial défini par $C_r^i = \frac{r!}{i!(r-i)!}$; C_r^i correspond aussi au nombre de façons de choisir i éléments parmi r.

1. Pour a et b éléments d'un anneau qui commutent, la formule du binôme donne

$$(a+b)^r = \sum_{i=0}^{r} C_r^i a^i b^{r-i}$$

On déduit la première égalité pour $a = b = 1$.
On rappelle que

$$1 + x + x^2 + \ldots + x^r = \frac{x^{r+1} - 1}{x - 1}$$

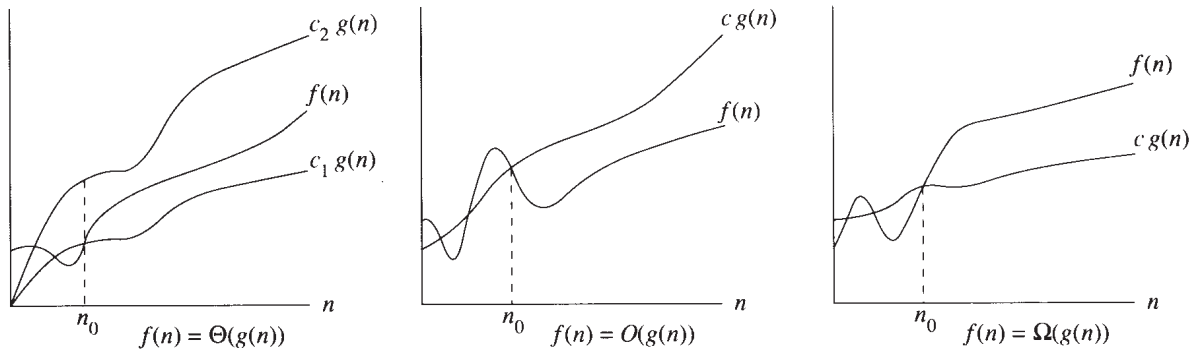
En effet

$$(x-1)\sum_{i=0}^{r} x \;\; = \;\; \sum_{i=1}^{r+1} x - \sum_{i=0}^{r} x = x^{r+1} - 1$$

Avec $x = 1$ on obtient la deuxième égalité.

2. Noter que par abus, on note souvent $f(n) = O(g(n))$ au lieu de $f \in O(g)$.

$$\begin{aligned}
f(n) = O(g(n)) \Leftrightarrow & \text{ il existe des constantes stt positives } c \text{ et } n_0 \\
& \text{telles que } 0 \leq f(n) < c.g(n) \text{ pour tout } n \geq n_0 \\
\Leftrightarrow & \limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty \\
f(n) = \Omega(g(n)) \Leftrightarrow & \text{ il existe des constantes stt positives } c \text{ et } n_0 \\
& \text{telles que } 0 \leq c.g(n) \leq f(n) \text{ pour tout } n \geq n_0 \\
\Leftrightarrow & \liminf_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0 \\
f(n) = \Theta(g(n)) \Leftrightarrow & \text{ il existe des constantes stt positives } c_1,\, c_2 \text{ et un } n_0 \\
& \text{tels que } 0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n) \text{ pour tout } n \geq n_0
\end{aligned}$$



Dans le cas où $\frac{f(n)}{g(n)}$ admet une limite à l'infini, on a

— $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = O(g(n))$

— $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \Rightarrow f(n) = \Theta(g(n))$

3. On sait qu'il existe $c > 0$ et n_0 tq $0 \leq f(n) \leq c.g(n)$ pour $n \geq n_0$.
Soit $f' \in O(f)$. Il existe $c' > 0$ et n_0' tq $0 \leq f'(n) \leq c'.f(n)$ donc pour $n \geq \max(n_0, n_0')$ on a $0 \leq f'(n) \leq cc'g(n)$.
En particulier on a $O(f) \subset O(g)$ lorsque $f \leq g$.
Posons $h = \max(f, g)$. Pour $n \geq n_0$ on a donc $g(n) \leq h(n) \leq \max(1, c).g(n)$. On en déduit les inclusions $O(g) \subset O(h) \subset O(g)$.

4. Se déduit de la question précédente an remarquant que $h \leq f + g \leq 2h$.

5. 1) $\sum_{p=1}^{n} p^k \leq n.n^k = n^{k+1}$ donc $\sum_{p=1}^{n} p^k \in O(n^{k+1})$.

2) nouvelle présentation (2018)

Soit m la partie entière de $n/2$. Notons que $n-m \geq n/2$. En minorant par $n/2$ les $n-m$ entiers entre $n/2$ et n on obtient

$$\begin{array}{rcl} (n/2)^k & \leq & (m+1)^k, \\ (n/2)^k & \leq & (m+2)^k, \\ & \cdots & \\ (n/2)^k & \leq & n^k \end{array}$$

En sommant ces $n-m$ inégalités

$$(n/2)(n/2)^k \leq (n-m)\,(n/2)^k \leq \sum_{p=m+1}^{n} p^k \leq \sum_{p=1}^{n} p^k$$

d'où

$$n^{k+1} \leq 2^{k+1} \sum_{p=1}^{n} p^k \ .$$

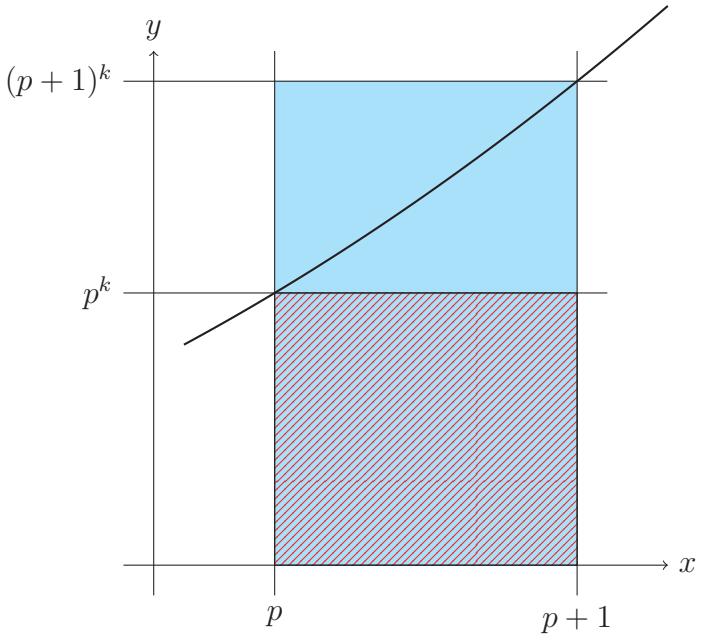
k étant fixé, 2^{k+1} est une constante donc $n^{k+1} \in O(\sum_{p=1}^{n} p^k)$.

solution alternative

La fonction $x \mapsto x^k$ est croissante, donc pour tout $x \in [p; p+1]$, on a

$$p^k \leq x^k \leq (p+1)^k$$

Sur un dession cela donne



En rappelant que l'intégrale calcule l'aire sous la courbe (et on intégrant) on obtient :

$$p^k \leq \int_{p}^{p+1} x^k dx \leq (p+1)^k$$

Ainsi, en sommant l'inégalité à gauche de 1 à n et l'inégalité à droite de 0 à $n-1$, on obtient :

$$\int_{0}^{n} x^k dx \leq \sum_{p=1}^{n} p^k \leq \int_{1}^{n+1} x^k dx$$

$$\frac{n^{k+1}}{k+1} \leq \sum_{p=1}^{n} p^k \leq \frac{(n+1)^{k+1}-1}{k+1}$$

Ainsi

$$\sum_{p=0}^{n} p^k \sim \frac{n^{k+1}}{k+1}$$

ancienne présentation 1

Pour montrer $n^{k+1} \in O(\sum_{p=1}^{n} p^k)$, on peut faire apparaître une puissance k à l'aide la formule du binôme en prenant $a = n-1$ et $b = 1$:

$$n^{k+1} = \sum_{i=0}^{k+1} C_{k+1}^i (n-1)^i \, 1^{k+1-i} = \sum_{i=0}^{k} C_{k+1}^i (n-1)^i + (n-1)^{k+1}$$

Dans la somme on majore chaque $(n-1)^i$ par $(n-1)^k$, ce qui donne

$$n^{k+1} \leq (n-1)^{k+1} + (n-1)^k \sum_{i=0}^{k} C_{k+1}^i \leq (n-1)^{k+1} + (n-1)^k \, 2^{k+1}$$

puisque $\sum_{i=0}^{k} C_{k+1}^i \leq \sum_{i=0}^{k+1} C_{k+1}^i = 2^{k+1}$. On en déduit la suite d'inégalités

$$\begin{array}{rcl} n^{k+1} & \leq & (n-1)^{k+1} + 2^{k+1} \, (n-1)^k \\ (n-1)^{k+1} & \leq & (n-2)^{k+1} + 2^{k+1} \, (n-2)^k \\ & \cdots & \\ 3^{k+1} & \leq & 2^{k+1} + \; 2^{k+1} \, 2^k \\ 2^{k+1} & \leq & 1^{k+1} + \; 2^{k+1} \, 1^k \end{array}$$

En les sommant de part et d'autre il apparaît

$$n^{k+1} \leq 1 + 2^{k+1} \sum_{p=1}^{n-1} p^k \leq 2^{k+1} \sum_{p=1}^{n} p^k$$

k étant fixé, 2^{k+1} est une constante donc $n^{k+1} \in O(\sum_{p=1}^{n} p^k)$.

ancienne présentation 2

Pour montrer que $n^{k+1} \in O(\sum_{p=1}^{n} p^k)$ on exprime n^{k+1} en fonction des p^k, en suivant la même idée que lorsqu'on calcule $\sum_{p=1}^{n} p^k$: on développe $\sum_{p=1}^{n} (p+1)^{k+1}$.

$$\begin{array}{rcl} (p+1)^{k+1} & = & \displaystyle\sum_{j=0}^{k+1} C_{k+1}^j p^j = p^{k+1} + \sum_{j=0}^{k} C_{k+1}^j p^j \\ \displaystyle\sum_{p=1}^{n-1} (p+1)^{k+1} & = & \displaystyle\sum_{p=1}^{n-1} p^{k+1} + \sum_{p=1}^{n-1} \sum_{j=0}^{k} C_{k+1}^j p^j \end{array}$$

Par soustraction on obtient :

$$\begin{aligned} n^{k+1} - 1 &= \sum_{p=1}^{n-1} \sum_{j=0}^{k} C_{k+1}^{j} p^j \\ n^{k+1} &\leq 1 + \sum_{p=1}^{n-1} \sum_{j=0}^{k} C_{k+1}^{j} p^k \\ n^{k+1} &\leq 1 + \sum_{p=1}^{n-1} p^k \sum_{j=0}^{k} C_{k+1}^{j} \\ n^{k+1} &\leq \sum_{p=1}^{n-1} p^k \sum_{j=0}^{k+1} C_{k+1}^{j} \\ n^{k+1} &\leq (\sum_{p=1}^{n-1} p^k) 2^{k+1} \end{aligned}$$

k étant fixé, 2^{k+1} est une constante donc $n^{k+1} \in O(\sum_{p=1}^{n} p^k)$.

Exercice 1.2 ***: Complexité dans le pire des cas***

Question 1.2.1 Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur *certaines* données ?

Question 1.2.2 Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur *toutes* les données ?

Question 1.2.3 Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur *certaines* données ?

Question 1.2.4 Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur *toutes* les données ?

Solution

1. Oui

2. Oui

3. Oui

4. Non

Exercice 1.3 ***: Classement de fonctions***

Question 1.3.1 Sur une échelle croissante, classer les fonctions suivantes selon leur comportement asymptotique : c'est-à-dire $g(n)$ suit $f(n)$ si $f(n) = O(g(n))$.

$f_1(n) = 2n$	$f_2(n) = 2^n$	$f_3(n) = \log(n)$	$f_4(n) = \frac{n^3}{3}$
$f_5(n) = n!$	$f_6(n) = \log(n)^2$	$f_7(n) = n^n$	$f_8(n) = n^2$
$f_9(n) = n + \log(n)$	$f_{10}(n) = \sqrt{n}$	$f_{11}(n) = \log(n^2)$	$f_{12}(n) = e^n$
$f_{13}(n) = n$	$f_{14}(n) = \sqrt{\log(n)}$	$f_{15}(n) = 2^{\log_2(n)}$	$f_{16}(n) = n\ \log(n)$

Remarque :

$$\log_a n = \frac{\log_b n}{\log_b a} \qquad\qquad n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Solution

On classe d'abord les fonctions en trois catégories : les fonctions polylogarithmiques ($\log^k n$), les fonctions polynomiales (n^k) et les fonctions exponentielles (a^n). Pour les autres fonctions, on essaie quand même de les faire rentrer dans une de ces trois catégories. Par exemple pour $n \geq 2$, on a $2^n \leq n! \leq n^n$ donc $n!$ est « exponentielle ».

$$f_{14} < f_3 \equiv f_{11} < f_6 < f_{10} < f_{13} = f_{15} \equiv f_9 \equiv f_1 < f_{16} < f_8 < f_4 < f_2 < f_{12} < f_5 < f_7$$

$$\sqrt{\log(n)} < \log(n) \equiv \log(n^2) < \log(n)^2 < \sqrt{n} < n = 2^{\log_2(n)} \equiv n + \log(n) \equiv 2\,n$$
$$< n\,\log(n) < n^2 < \frac{n^3}{3} < 2^n < e^n < n! < n^n$$

2 Coût amorti

Dans l'analyse de *structure de données*, on s'intéresse à des *séquences* d'opérations. Étudier le coût pire-cas séparé de chacune des opérations est souvent trop pessimiste, car cela ne tient pas compte des répercussions que chaque opération a sur la structure de données. On s'intéresse donc au coût amorti d'une suite de n opérations, qui est *le coût moyen d'une opération dans le pire des cas*, et l'on se sert pour cela de deux méthodes :

- **la méthode par agrégat** consiste à *majorer* le coût de toute suite de n opérations effectuées sur la structure, et à diviser le coût total ainsi obtenu par le nombre n d'opérations ;
- **la méthode du potentiel** consiste à introduire une fonction $\phi(D_i)$ qui associe à l'état D_i de la structure de données après la i-ième opération un nombre réel ; le coût amorti $\hat{c}_i$ de la i-ième opération est alors le coût réel c_i, moins la différence de potentiel dûe à l'opération

$$\hat{c}_i = c_i + \phi\left(D_i\right) - \phi\left(D_{i-1}\right) ; \tag{1}$$

ainsi (puisque les $\phi(D_i)$ s'annulent deux à deux dans la somme), le coût amorti total est

$$\sum_{i=1}^{n} \hat{c}_i = \left(\sum_{i=1}^{n} c_i\right) + \phi(D_n) - \phi(D_0). \tag{2}$$

Principe

Le coût amorti d'une suite de n opérations est le coût moyen d'une opération dans le pire des cas. Cela a pour but d'atténuer le coût des opérations coûteuses en tenant compte du faible coût des autres opérations (piles) ou que l'utilisation de ces opérations coûteuses permet pour les autres opérations (files de Fibonacci). Tout en ayant une opération coûteuse on peut avoir un coût amorti faible !

Le calcul du coût amorti d'une opération en utilisant la *méthode par agrégat* est une méthode introduite par Aho et al. ("*The Design and Analysis of Algorithms*", 1974) qui précéde la notion

moderne d'analyse amortie formalisée dans un article de Tarjan en 1985 : selon cet article, l'introduction de la *méthode du banquier/comptable* (non traitée ici, car elle très proche de la méthode du potentiel) date d'un article de Brown et Tarjan, et est ensuite rafinée par Huddleston et Melhorn ; enfin, Tarjan attribue la *méthode du potentiel* à Sleator.

La méthode du potentiel est la plus compliquée à mettre en oeuvre, car elle nécessite de choisir un état à observer (nombre d'éléments contenus dans la structure, nombre d'éléments supprimés depuis la création de la structure, profondeur de la structure arborescente, etc.) qui est de plus *indicatif de l'efficacité de la structure pour les opérations à venir.*

Intuitivement, une opération peu coûteuse, mais qui cependant dégrade un peu l'efficacité de la structure (par exemple une insertion) doit faire augmenter un peu le potentiel en contre-partie ; une opération très coûteuse, qui augmente l'efficacité de la structure (par exemple, dans le cas des files de Fibonacci, la consolidation) doit faire fortement diminuer le potentiel.

Plus précisément on a une structure de données initiale D_0 sur laquelle on effectue n opérations. Pour $i = 1 \dots n$, on note c_i le coût réel de la i-ème opération et D_i la structure de données qui résulte de la i-ème opération sur la structure D_{i-1}. On se dote d'une fonction de potentiel ϕ : à chaque D_i, on associe un réel $\phi(D_i)$ qui est le *potentiel* associé à D_i. Le *coût amorti* de la i-ème opération $\hat{c}_i$ est défini par

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

c'est-à-dire la somme du coût réel et de la différence de potentiel dû à l'opération. Il suit que le coût amorti total est

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

et donc $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ si $\phi(D_n) \geq \phi(D_0)$. La plupart du temps on aura $\phi(D_0) = 0$, et on peut d'ailleurs toujours se ramener à ce cas en posant $\phi'(D) = \phi(D) - \phi(D_0)$. De plus, on vérifiera que pour tout $i \in \mathbb{N}, \phi(D_i) \geq \phi(D_0)$ pour que le coût amorti total soit un majorant du coût réel et que son calcul ait un sens (il peut aussi avoir un sens lorsque la différence de potentiel est négative—dans la méthode du banquier/comptable, on parle alors d'*emprunt*—mais par simplicité, on ignorera ce cas).

De nombreux exercices considèrent qu'au final on peut toujours multiplier la fonction de potentiel par une constante positive pour négliger un terme dans un grand-O (on appelle cela « *mise à l'échelle* », ou "*scaling*").

Exercice 2.1 ***: Coût amorti (piles avec multi-dépilement)***

On considère les deux opérations fondamentales sur les piles, `Empiler(S,x)` et `Dépiler(S)`, auxquelles on ajoute l'opération `MultiDépiler(S,k)` qui retire les k premiers objets du sommet de la pile si la pile contient au moins k objets et vide la pile sinon.

Question 2.1.1 Calculer le coût amorti d'une opération en utilisant la méthode par agrégat.

Question 2.1.2 Même question en utilisant la méthode du potentiel.

Solution

1. Les opérations `Empiler` et `Dépiler` se font en $O(1)$.
Pour contraster avec l'analyse amortie, on fait l'analyse pire-cas de l'opération `MultiDépiler` : son coût est $\min(|S|, k)$, où $|S|$ est le nombre d'éléments contenu dans la pile ; au pire, cette opération dépile donc *tous les éléments* de la pile ; et au pire, la pile est complètement remplie et contient n éléments. Ainsi l'opération se fait en $O(n)$ pire-cas.

Cette estimation est effectivement *plus qu'improbable* (puisqu'on ne peut empiler qu'au plus un élément par opération, on ne peut dépiler $O(n)$ élément qu'un nombre constant de fois lors de la suite d'opérations), mais d'une part, pour une analyse pire-cas il est fréquent (et souvent fructueux) de grossièrement surestimer la complexité ; d'autre part, c'est cette surestimation (ou plutôt l'argument qui justifie que c'est une surestimation) qui motive l'analyse amortie.

On remarque donc que le nombre d'appels à `Dépiler` (y compris ceux qu'on trouve dans `MultiDépiler`) est inférieur ou égal au nombre d'appels à `Empiler`. Ainsi, une séquence de n opérations `Empiler`, `Dépiler` et `MultiDépiler` coûtera au plus $O(n)$ opérations, ce qui fait un coût amorti en $O(1)$.

2. Le nombre d'objets dans la pile apparaît comme une bonne fonction de potentiel : la variation de potentiel pour les opérations peu coûteuses, `Empiler` et `Dépiler`, est faible et celle pour l'opération coûteuse `MultiDépiler` est négative et importante.

On a $\phi(D_0) = 0$ et $\phi(D_i) \geq 0$ pour tout i, si bien que $\sum_{i=1}^{n} \hat{c}_i$ majore le coût réel.

On considère la i-ième opération sur une pile contenant s objets.

(a) Si cette opération est `Empiler` alors $\phi(D_i) - \phi(D_{i-1}) = (s+1) - s = 1$ et $c_i = 1$, donc $\hat{c}_i = 2$.

(b) Si cette opération est `Dépiler` alors $\phi(D_i) - \phi(D_{i-1}) = -1$, $c_i = 1$ et $\hat{c}_i = 0$.

(c) Si cette opération est `MultiDépiler` alors on enlève $k' = \min(s, k)$ objets dans la pile, d'où $\phi(D_i) - \phi(D_{i-1}) = -k'$. `MultiDépiler` dépile k' éléments donc le coût réel de `MultiDépiler` est k' fois le coût de `Dépiler`, c'est-à-dire que $c_i = k'$, d'où $\hat{c}_i = k' - k' = 0$.

Ainsi $\sum_{i=1}^{n} \hat{c}_i \leq 2n$. Le coût amorti total est $O(n)$ et le coût amorti d'une opération est $O(1)$.

Exercice* 2.2 *: Coût amorti (piles avec multi-dépilement et multi-empilement)

Aux trois opérations précédentes, `Empiler(S,x)`, `Dépiler(S)` et `MultiDépiler(S,k)`, définies sur les piles, on ajoute l'opération `MultiEmpiler(S,`x_1`,`x_1`,`x_2`,...,`x_k`)`, qui permet d'empiler les objets $x_1, x_2, \ldots, x_k$ sur la pile. On distingue deux implantations différentes de cette pile qui induisent chacune une restriction supplémentaire sur l'opération `MultiEmpiler` :

(a) le nombre d'objets que l'on peut empiler en une opération est borné par une constante c ;

(b) le nombre d'objets que l'on peut empiler à chaque fois est inférieur ou égal au nombre d'objets de la pile S, noté $|S|$.

Il s'agit de calculer le coût amorti d'une opération dans le cas (a), puis dans le cas (b).

Remarque

La première version correspond à l'implantation d'une pile dans un bloc de mémoire ou un tableau de taille fixe (une métaphore possible, suggérée par le Cormen et al., est celle de la pile d'assiettes dans un placard non-extensible).

Il s'agit, dans la deuxième version d'une version dynamique de la première version, où une réallocation de la mémoire (ou du tableau) doit au préalable être faite ; c'est par ailleurs exactement ce qui est réalisé dans l'exercice suivant sur les tableaux dynamiques, sauf qu'ici l'heuristique est de prévoir qu'à chaque étape le tableau peut au plus doubler sa taille actuelle, pour que l'ordre de grandeur du tableau reste le même (l'heuristique dans l'exercice suivant prend en compte que le tableau ne croit que d'un élément par opération, et qu'il suffit de le doubler lorsqu'il est presque rempli).

Question 2.2.1 En utilisant la méthode par agrégat.

Question 2.2.2 En utilisant la méthode du potentiel.

Solution

1. On observe d'abord que dans les deux cas, la séquence la plus couteuse consiste à ne faire que des multi-empilements, en rajoutant le plus d'éléments possible. Plus précisément, si un dépilement est suivi d'un empilement alors on peut inverser les deux opérations en conservant le coût. La suite obtenue est bien définie. Le coût d'une suite est donc identique à celui de la suite obtenue en faisant glisser tous les empilements au début.

(a) La taille de la pile est au plus cn, donc on ne peut pas dépiler plus de cn objets. Le coût total est au plus $2cn$, et le coût amorti est donc $O(1)$.

(b) Au pire : $1 + 2^0 + \cdots + 2^i + \cdots + 2^{n-2} = 2^{n-1}$. Le coût amorti est $O(2^{n-1}/n)$.

2. On reprend la fonction de potentiel ϕ utilisée précédemment, le nombre d'objets contenus dans la pile.

(a) Lorsqu'on empile, on a $c_i \leq c$ et $\phi(D_i) - \phi(D_{i-1}) \leq c$, si bien que $\hat{c}_i \leq 2c$.
Lorsqu'on dépile $c_i = -\phi(D_{i-1}) - \phi(D_i)$ donc $\hat{c}_i = 0$.
Dans tous les cas $\hat{c}_i \leq 2c$ et on retrouve un coût amorti en $O(1)$.

(b) $c_i = k'$ avec $k' \leq 2^{i-1}$, et $\phi(D_i) - \phi(D_{i-1}) = k'$, d'où $\hat{c}_i = 2k' \leq 2^i$.

Exercice 2.3 ***: Coût amorti (tableaux dynamiques)***

Remarque

Si l'on préfère, on peut traiter la question 3 (qui illustre le problème) avant les autres. D'autre part, on peut faire noter rapidement—soit lors de la correction de l'exercice, soit après le cours sur le hachage—que l'esprit de la technique donnée ici est central aux tables de hachage, pour lesquelles il permet d'obtenir l'accès en temps constant $O(1)$. Dans les tables de hachage cependant, on utilise un mécanisme plus flexible permettant de fixer le taux de remplissage qui déclenche le doublement du tableau (ici, il est fixe).

Le but de cet exercice est d'illustrer la recherche d'une fonction de potentiel, dont le choix apparaît aux étudiants comme étant arbitraire.

Une séquence de n opérations est effectuée sur une structure de données. La i-ème opération coûte i si i est une puissance de 2 et 1 sinon.

Question 2.3.1 En utilisant la méthode par agrégat, montrer que le coût amorti par opération est en $O(1)$.

Question 2.3.2 Méthode du potentiel.

Idée : le potentiel croît pour chaque opération peu coûteuse (*i.e.* lorsque i n'est pas une puissance de 2) et retombe à 0 pour chaque opération coûteuse (*i.e.* lorsque i est une puissance de 2).

(a) *Essai 1.* Le potentiel croît de 1 pour chaque opération peu coûteuse. Autrement dit :

$$\phi(i) = \begin{cases} 0 & \text{si } i = 0 \\ \phi(i-1) + 1 & \text{si } i \text{ n'est pas une puissance de 2} \\ 0 & \text{si } i \text{ est une puissance de 2} \end{cases}$$

Calculer le coût amorti en utilisant cette fonction de potentiel. Le résultat est-il satisfaisant ?

(b) *Essai 2.* Le potentiel croît de 2 pour chaque opération peu coûteuse. Autrement dit :

$$\phi(i) = \begin{cases} 0 & \text{si } i = 0 \\ \phi(i-1) + 2 & \text{si } i \text{ n'est pas une puissance de 2} \\ 0 & \text{si } i \text{ est une puissance de 2} \end{cases}$$

Calculer le coût amorti en utilisant cette fonction de potentiel. Le résultat est-il satisfaisant ?

Question 2.3.3 Donner un exemple d'une suite d'opérations ayant comme coût le coût décrit dans cet exercice (préciser la structure de données).

Chercher éventuellement un exemple de coût moyen facile a calculer puis faire un coût amorti qu'on espère différent.

Solution

1. Soit k l'entier naturel tel que $2^k \leq n < 2^{k+1}$. Autrement dit k est la partie entière de $\log_2(n)$. Soient c_i le coût de la i-ème opération et c le coût total de la suite des n opérations, alors :

$$\begin{aligned} c &= \underbrace{\sum_{1\leq i\leq n} c_i}_{i \text{ puissance de } 2} + \underbrace{\sum_{1\leq i\leq n} c_i}_{i \text{ non puissance de } 2} \\ &= \sum_{j=0}^{k} 2^j + n - (k+1) \\ &= (2^{k+1}-1) + n - (k+1) \\ &\leq 3n - k - 2 \quad (\text{puisque } 2^k \leq n \text{ on a } 2^{k+1} \leq 2n) \\ &\leq 3n \end{aligned}$$

Le coût amorti est égal à c/n et est donc en $O(1)$.

2. (a) Le tableau suivant montre les valeurs du potentiel, de la différence de potentiel, du coût réel et du coût amorti pour chaque opération i.

i	0	1	2	3	4	5	6	7	8	9	10	...	15	16	...	2^j+1	...	$2^{j+1}-1$	2^{j+1}
$\phi(i)$	0	0	0	1	0	1	2	3	0	1	2	...	7	0	...	1	...	2^j-1	0
ddp		0	0	1	-1	1	1	1	-3	1	1	...	1	-7	...	1	...	1	$-(2^j-1)$
c_i		1	2	1	4	1	1	1	8	1	1	...	1	16	...	1	...	1	2^{j+1}
$\hat{c_i}$		1	2	2	3	2	2	2	5	2	2	...	2	9	...	2	...	2	2^j+1

On obtient un coût amorti $\hat{c_i}$ qui est tantôt 2 tantôt $i/2+1$. Ce n'est pas très satisfaisant car il faudrait refaire une moyenne des n coûts amortis pour arriver à un coût amorti moyen en $O(1)$.
(b) Le même tableau, pour l'autre fonction de potentiel.

i	0	1	2	3	4	5	6	7	8	9	10	...	15	16	...	2^j+1	...	$2^{j+1}-1$	2^{j+1}
$\phi(i)$	0	0	0	2	0	2	4	6	0	2	4	...	14	0	...	2	...	$2^{j+1}-2$	0
ddp		0	0	2	-2	2	2	2	-6	2	2	...	2	-14	...	2	...	2	$-(2^{j+1}-2)$
c_i		1	2	1	4	1	1	1	8	1	1	...	1	16	...	1	...	1	2^{j+1}
$\hat{c_i}$		1	2	3	2	3	3	3	2	3	3	...	3	2	...	3	...	3	2

On obtient un coût amorti $\hat{c_i}$ qui est tantôt 2 tantôt 3, donc toujours en $O(1)$, ce qui est satisfaisant.

3. La structure de données est une structure de tableau « dynamique » dont la taille i double à chaque fois qu'il est presque plein, c'est-à-dire dès que l'ajout d'un élément le remplirait. On néglige le coût de l'allocation. Le coût est donc constitué de la recopie éventuelle des (i-1) éléments et de l'ajout du i-ième élément dans le nouveau tableau. On démarre avec un tableau de taille 1.

— Insertion 1 : le tableau est presque plein donc on double sa taille et on y met un élément.
— Insertion 2 : le tableau a une taille 2 et contient un élément x_1 ; il est presque plein donc on crée un tableau de taille double (donc de taille 4), on y recopie x_1 et on y insère le deuxième élément.
— Insertion 3 : le tableau a une taille 4 et contient deux éléments, on y insère le troisième élément.

— Insertion 4 : le tableau a une taille 4 et contient trois éléments x_1, x_2, x_3. Il est presque plein ; on crée un tableau de taille double (donc de taille 8), on y recopie x_1, x_2, x_3 et on y insère le quatrième élément.

Et ainsi de suite ...

— Insertion $i = 2^k$: le tableau a une taille $i = 2^k$ et contient $i - 1$ éléments $x_1, x_2, \ldots, x_{i-1}$, on prévoit qu'après la i-ième insertion il sera plein, on crée un tableau de taille double (donc de taille 2^{k+1}), on y recopie $x_1, x_2, \ldots, x_{i-1}$ et on y insère le i-ième élément.

Ici le potentiel défini dans l'essai 2 vaut *deux fois le nombre de cases occupées dans la seconde moitié du tableau* ; ce qui justifie le choix de la fonction (car il s'agit bien d'un état de la structure de données).

Exercice* 2.4 *: Coût amorti (incrémentation d'un compteur)

Soit k un entier positif. On considère les entiers naturels inférieur à 2^k. Étant donné un tel entier x, son écriture en binaire peut être stockée dans un tableau $A[0..k-1]$ ne contenant que des 0 et des 1 (le bit de poids le plus faible est stocké dans $A[0]$). On définit l'opération `incrementer(A)` comme suit :

```
Incrementer(A)
  i <- 0 ;
  tantque i < k et A[i] = 1 faire
    A[i] <- 0
    i <- i+1
  fintantque
  si i < k alors
    A[i] <- 1
  finsi
fin
```

Question 2.4.1 Quel est l'effet de `Incrementer` sur le tableau A ?

Question 2.4.2 Le coût réel de l'opération `Incrementer` est le nombre de bits qui changent. On considère une suite de n incrémentations à partir de 0 (on suppose $n < 2^k$). Calculer le coût amorti de l'opération `Incrementer` :

(a) en utilisant la méthode par agrégat ;

(b) en utilisant la méthode du potentiel (une fois la fonction de potentiel choisie, il est conseillé de faire un tableau " *coût réel/variation de potentiel/coût amorti* " pour les premières incrémentations d'un compteur à 4 bits).

Solution

1. Par exemple, pour $A = [1, 1, 0, 1]$, on obtient : $[0, 0, 1, 1]$. L'effet de `Incrémenter` est donc simple à décrire : on fait basculer à 0 les premiers bits égaux à 1, et on met à 1 le bit suivant le dernier traité.

2. (a) Le premier bit bascule à chaque fois, le second une fois sur deux, le troisième une fois sur quatre, etc. Ainsi, le nombre de changements de bits pour $2^{i-1} \leq n < 2^i$ est

$$n + n/2 + n/2^2 + \cdots + n/2^i \leq 2n$$

si bien que le coût amorti est $O(1)$. Dans un tableau représentant les états successifs du compteur sur les lignes, on visualise bien le nombre de changements du $i$ème bit dans la colonne i. Les nombres de changements peut aussi être lus d'une ligne à l'autre mais on ne voit pas apparaître clairement une régularité $(1, 2, 1, 3, 1, 2, 1, 4)$

(b) Le potentiel considéré est *le nombre de* 1 *contenus dans le tableau* A (intuitivement, plus il y a de 1 dans le tableau, plus il y a de chance que l'opération `Incrementer` soit coûteuse). Le coût amorti $\hat{c}_i$ de la i-ième opération, lorsqu'on a posé $\Delta_i := \phi(A_i) - \phi(A_{i-1})$, est

$$\hat{c}_i = c_i + \Delta_i.$$

On note respectivement t_{01} et t_{10} le nombre de bits qui passent de 0 à 1 et de 1 à 0 au cours de la i-i-ème opération. Les observations importantes sont les suivantes :
— le coût réel c_i est, comme nous l'avons déjà observé, le nombre de bits changés, $c_i = t_{01} + t_{10}$;
— la différence de potentiel est la variation du nombre de bits à 1, $\Delta_i = t_{01} - t_{10}$;
— enfin, à chaque incrémentation, on ne change qu'un bit de 0 à 1 (le premier bit à 0 lu), $t_{01} = 1$.
Cela permet de conclure au coût amorti $\hat{c}_i = 2$, prouvant que l'incrémentation a un coût amorti en $O(1)$.

3 Files binomiales

Exercice 3.1 ***: Définitions et propriétés des arbres binomiaux***
On rappelle deux définitions équivalentes des *arbres binomiaux*, introduits dans le premier cours.

Remarque

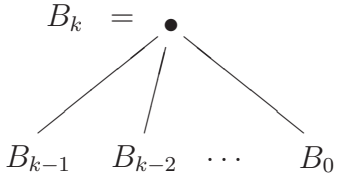
Pendant la séance, penser à dessiner des arbres binomiaux. De plus, il peut être judicieux de remarquer que les arbres peuvent être dessinés de deux façons différentes : en fonction de si l'on cherche à mettre en avant la première définition (dans ce cas, pour dessiner B_k, on dessine côte-à-côte B_0, B_1, …, B_{k-1}, puis on rajoute un nœud-racine qui a pour fils ces k arbres), ou la seconde (pour dessiner B_k, on dessine deux B_{k-1} l'un à côté de l'autre, à hauteur légèrement décalée, et on fait l'un fils de l'autre).

Il existe d'autres définitions des arbres binomiaux, dont en voici une très *jolie* qui peut être mentionnée au détour de l'exercice : l'arbre binomial B_0 est réduit à un nœud ; ensuite, B_{k+1} est la *couronne* de B_k. La *couronne* ("*corona*") d'un graphe est obtenue en rajoutant une feuille à chaque nœud du graphe—le graphe obtenu comporte le double de nœuds.

Définition 1.
— B_0 est l'arbre réduit à un seul nœud ;
— étant donnés deux arbres binomiaux B_{k-1}, on obtient B_k en faisant de l'un des B_{k-1} le premier fils à la racine de l'autre B_{k-1}.

Définition 2.
— B_0 est l'arbre réduit à un seul nœud ;
— B_k est l'arbre dont la racine à k fils : B_{k-1}, ..., B_0.

$$B_0 = \bullet$$
$$B_k = (\bullet, B_{k-1}, \ldots, B_0)$$



Question 3.1.1 Prouver d'abord que ces deux définitions sont équivalentes. (noter $\mathcal{D}_i$ l'ensemble défini inductivement par la définition i et montrer que $\mathcal{D}_1 \subset \mathcal{D}_2$ et $\mathcal{D}_2 \subset \mathcal{D}_1$).

Question 3.1.2 Montrer ensuite chacune de ces propriétés portant sur les arbres binomiaux en choisissant à chaque fois la définition qui semble la plus pratique :

(a) B_k a 2^k nœuds ;

(b) la hauteur de B_k est k ;

(c) il y a C_k^i nœuds à la profondeur i $(i = 0, \ldots, k)$;

(d) la racine de B_k est de degré k, supérieur au degré de n'importe quel autre nœud de B_k.

Solution

1. *En se plaçant sous la définition 2.* Soit $B_k \in \mathcal{D}_2$ un arbre avec k fils B_0, ..., B_{k-2}, B_{k-1} appartenant tous à $\mathcal{D}_2$. Supposons qu'ils satisfont à l'hypothèse d'induction. L'arbre B obtenu en retirant le B_{k-1} de cet arbre est formé d'une racine à $k-1$ fils B_0, ..., B_{k-2}. Selon la définition 2, B est donc aussi un B_{k-1}. Par hypothèse de récurrence B et B_{k-1} sont dans $\mathcal{D}_1$. Ainsi liés il forment un B_k suivant la définition 1. Par conséquent $B_k \in \mathcal{D}_1$.

En se plaçant sous la définition 1. Soit $B_k \in \mathcal{D}_1$ un arbre formé de la liaison ("*linking*") de deux $B_{k-1} \in \mathcal{D}_1$. On note B'_{k-1} celui qui porte la racine de B_k. Par hypothèse de récurrence, B_{k-1} et B'_{k-1} sont dans $\mathcal{D}_2$. B'_{k-1} a donc $k-1$ fils B_0, ..., B_{k-2}, tous dans $\mathcal{D}_2$. D'après la définition 2, B_k appartient à $\mathcal{D}_2$.

2. (a) Par récurrence sur k. Dans l'étape inductive, si on prend la première définition, on utilise que $2^{k-1} + 2^{k-1} = 2^k$; si on prend la seconde définition, on utilise la somme géométrique,

$$(1 + 2 + \ldots + 2^{k-1}) + 1 = (2^k - 1) + 1 = 2^k$$

qui représente la taille des fils, auxquelles on rajoute la racine de l'arbre.

(b) Avec les deux définitions, la hauteur de B_k est trivialement un de plus que la hauteur de B_{k-1}. Par récurrence, on a donc une hauteur de k.

(c) On note $N(k, i)$ le nombre de nœuds de B_k à profondeur i pour une récurrence sur i. Le cas de base est $N(k, 0) = 1$ (puisqu'il n'y a jamais qu'une racine dans un arbre). En utilisant la première définition, on a

$$\begin{aligned} N(k,i) &= N(k-1,i) + N(k-1,i-1) \\ &= C_{k-1}^i + C_{k-1}^{i-1} \\ &= C_k^i \end{aligned}$$

grâce à l'hypothèse de récurrence et au triangle de Pascal.

La seconde définition donne lieu à la récurrence suivante

$$\begin{aligned} N(k,i) &= N(0,i-1) + \ldots + N(k-1,i-1) \\ &= \sum_{j=0}^{k-1} N(j,i-1) \end{aligned}$$

et on s'arrange pour retomber sur la récurrence précédente (on choisit de soustraire par $N(k-1,i)$ plutôt que $N(k-1,i-1)$, car le paramètre i ne varie pas dans la somme, et les termes ne s'annulent alors pas),

$$N(k,i) - N(k-1,i) = \sum_{j=0}^{k-1} N(j,i-1) - \sum_{j=0}^{k-2} N(j,i-1) = N(k-1,i-1).$$

(d) ***Rappel*** *: le degré d'un nœud est le nombre de ses fils ; le degré d'un arbre est le degré de sa racine.*
Induction avec la définition 2. D'après la définition la racine de B_k est de degré k. Tout autre noeud de B_k est un noeud d'un B_i où $i < k$. Par induction son degré est inférieur ou égal à i donc inférieur à k.

Conclusion. Comme la propriété (d) remarque que le degré (à la racine) de B_k est k, la propriété (a) permet de conclure que les arbres binomiaux ont *une taille exponentielle en leur degré* ; les propriétés (a) et (b) ensemble montrent que les arbres binomiaux ont *une hauteur logarithmique en leur taille.* Ces deux constatations expriment le caractère "bien équilibrés" de ces arbres.

Exercice 3.2 ***: Files binomiales relâchées***

Par définition, une file binomiale est composée de tournois binomiaux de tailles toutes différentes. À cause de cette contrainte sur les tailles, l'union de deux files binomiales de tailles respectives n et m a une complexité en $O(\log(n+m))$ et l'insertion d'un nouvel élément dans une file binomiale de taille n a une complexité en $O(\log(n))$. On souhaite maintenant avoir une structure plus souple dans laquelle on pourra réaliser l'union et l'insertion en $O(1)$, en reportant les opérations coûteuses sur la suppression du minimum. Cela est possible en supprimant la contrainte sur les tailles.

Question 3.2.1 Rappeler les tailles des arbres binomiaux qui forment une file binomiale de taille n, et les degrés des différentes racines. Quelle est la composition d'une file binomiale de taille 143 ?

Question 3.2.2 On pose $D(n) = \lfloor \log_2(n) \rfloor$.
Montrer que le degré maximum d'un nœud d'une file binomiale de taille n est $D(n)$.

On appelle *file binomiale relâchée* une suite de tournois binomiaux de tailles quelconques.
Un exemple de file binomiale relâchée : $(B_3, B_1, B_2, B_1, B_1, B_2, B_4, B_0, B_0, B_0)$.

Question 3.2.3 Montrer que le degré d'un nœud d'une file binomiale relâchée est au plus $D(n)$.

Étant donnés une file binomiale relâchée H et un nœud x quelconque de H :
- $\min(H)$ désigne la racine de l'arbre qui contient la plus petite clé (si H est vide alors $\min(H) = nil$)
- `taille` (H) désigne le nombre de nœuds dans la file H
- `pere` (x) est le père de x (si x est une racine alors `pere` $(x) = nil$)
- `fils` (x) est l'un des fils de x (si x est une feuille alors `fils` $(x) = nil$)
- `pred` (x) est le frère gauche de x et `succ` (x) est son frère droit (si x est une racine alors `pred` (x) et `succ` (x) sont des racines, si x est fils unique alors `pred` $(x) =$ `succ` $(x) = x$)
- `degre` (x) est le nombre de fils de x
- `clef` (x) est la valeur de la clé x

On peut s'appuyer sur une structure de données qui permet de lire et modifier ces valeurs en temps constants. De plus on dispose de la procédure `CréerFBRVide` de création d'une file binomiale relâchée vide.

Question 3.2.4 Écrire les procédures

(a) `ConcatenerFBR` d'union de deux files binomiales relâchées (en une file binomiale relâchée). Quelle est la complexité ?

(b) `InsererFBR` d'insertion d'une nouvelle valeur v dans une file binomiale relâchée. Quelle est la complexité ?

Question 3.2.5

(a) Écrire une procédure `Consolider` qui transforme une file binomiale relâchée en une file binomiale (avec contrainte sur les tailles).

(b) On note $a(H)$ le nombre d'arbres d'une file binomiale relâchée H.
Montrer que le coût de la procédure `Consolider` appliquée à une file H de taille n est inférieur ou égal à $\alpha(a(H) + D(n))$ où α est une constante (qui ne dépend pas de H).

Question 3.2.6 Écrire une procédure `ExtraireMinFBR` qui extrait le minimum d'une file binomiale relâchée non vide tout en la consolidant. La file obtenue après extraction du minimum est donc une file binomiale (avec contrainte sur les tailles).
S'assurer que $\min(H)$ est bien positionné sur la racine de plus petite clé de l'arbre obtenu après application de la procédure. Si nécessaire on pourra utiliser une procédure `MettreAJourMin` qui remet à jour $\min(H)$.

Question 3.2.7 En utilisant la méthode du potentiel, évaluer le coût amorti de la procédure `ExtraireMinFBR` d'extraction du minimum d'une file binomiale relâchée.
Indication : prendre une fonction de potentiel proportionnelle au nombre d'arbres de la file initiale.

Solution

Note : le pointer père ne semble servir qu'à tester si le noeud est une racine ou non.

1. On peut d'abord rappeler que pour tout entier n, il existe une file binomiale de taille n.
Soit $n = \sum_{k=0}^{D(n)} b_k 2^k$ et E_n l'ensemble des k tels que $b_k = 1$. Alors la file FB_n est composée des B_k tels que k est élément de E_n. La taille de ces arbres est 2^k et le degré de leur racine est k d'après l'ex. 3.1.

$$FB_{143} =< B_7, B_3, B_2, B_1, B_0 >$$

2. Le degré maximum d'un nœud de FB_n découle de l'ex 3.1.d puisque tout k de E_n vérifie $k \leq D(n)$.

3. Par l'absurde (valable aussi pour la question précédente). Supposons qu'il existe un nœud de degré $k > D(n)$. Puisque $D(n) \leq \log_2(n) < D(n) + 1 \leq k$ on a

$$2^{D(n)} \leq 2^n < 2^{D(n)+1} \leq 2^k \ .$$

D'après l'ex 3.1(a et d), le nœud serait dans un arbre B_k de taille 2^k plus grande que la taille de la file elle-même !

4. Chaque nœud x contient
— un pointeur vers son père
— un pointeur vers l'un de ses fils
les enfants de x sont reliés entre eux par une liste circulaire ; chaque enfant y de x a :
— un pointeur vers son frère gauche
— un pointeur vers son frère droit
Si y est fils unique, alors les deux pointeurs ci-dessus pointent sur y. Chaque nœud x contient aussi le nombre de ses fils. **Donner un exemple.**

(a) `ConcatenerFBR`
Pour réaliser l'union de deux files binomiales relâchées H et G, il suffit de concaténer les deux listes de racines et de tenir à jour le min.
Concaténation des listes de racines : c'est la concaténation de deux listes circulaires doublement chaînées. On pose $x = \min(H)$, $x' =$ `succ` (x), on insère la liste des racines de G entre x et x', en mettant à jour ceux des prédécesseurs et successeurs qui doivent être modifiés. On pose $y = \min(G)$, $y'' =$ `pred` (y), le nouveau successeur de x est y (et le nouveau prédécesseur de y est x), le nouveau prédécesseur de x' est y'' (le nouveau successeur de y'' est x').
Mise à jour du min : il suffit de calculer le minimum entre `clef` (x) et `clef` (y).
La complexité totale est en $O(1)$.

```
ConcatenerFBR(T1, T2)
  Concatener la liste des racines de T1 et T2 et tenir a jour le min.
```

(b) `InsererFBR`

```
InsererFBR(H, v)
 Creer un noeud x et une file binomiale G reduite a x :
      CreerFBRVide(G)
      degre(x)=0
      pere(x)=Nil
      fils(x)=Nil
      pred(x)=x
      succ(x)=x
      clef(x)=v
      min(G)=x
 ConcatenerFBR(H,G)
Fin InsererFBR
```

La complexité est en $O(1)$.

5. (a)

```
Consolider(H)
  n = taille(H)
  Si n > 0 Alors
    Creer un tableau A[0..D(n)] vide
    Pour Chaque noeud w de la liste des racines
      i = degre(w)
      TantQue A[i] non vide
            Si clef(w) > clef(A[i]) Alors echanger w et A[i]
            Faire de A[i] un fils de w
            A[i] = vide
            i = i+1
       Fin TantQue
       A[i] = w
       Mettre a jour les pointeurs
       Supprimer w de H
    Fin Pour                                  // H est vide
    Pour i allant de 0 jusqu'a D(n)
      Si A[i] non vide Alors
          Creer une file binomiale G reduite a A[i]
          ConcatenerFBR(H,G)
      Fin Si
    Fin pour
  Fin Si
Fin Consolider
```

Remarques :

— le nombre d'arbres de la file consolidée est inférieur ou égal au nombre d'arbres de la file initiale

— quand on supprime le nœud w de la liste des racines, on se contente de faire le raccord entre `pred` (w) et `succ` (w), sans remettre à jour le minimum. De toutes façons, lors des `ConcaténerFBR` il y aura, par comparaison successives des clés des racines, une mise à jour du minimum pendant qu'on reconstruit la file à partir du tableau.

— lorsqu'on consolide une file H, il est inutile d'imposer que $\min(H)$ contienne la clé minimum.

(b) Complexité c de `Consolider` (solution de Maryse, qui ne sait pas faire autrement) :

— on initialise un tableau de taille $D(n)+1$, coût $c_1 \leq \alpha_1 D(n)$

— on passe dans la boucle "`Pour Chaque noeud w...`", coût c_2 à calculer

— on crée une file binomiale à partir des (au plus) $D(n)+1$ arbres du tableau, coût $c_3 \leq \alpha_3 D(n)$

Majoration du coût c_2... au moyen d'un calcul de coût amorti (d'un tour de boucle). En effet le coût exact d'un tour est très variable. Par ailleurs lorsqu'il y a beaucoup de cases non vides dans le tableau A au début d'une étape, la possibilité de restructuration est forte.

On calcule le coût amorti d'une itération en prenant comme fonction de potentiel le nombre de cases non vides du tableau A multiplié par une constante λ, que l'on déterminera. On pose :

c_w = coût réel du traitement de w
$\hat{c}_w$ = coût amorti du traitement de w
S = structure avant traitement de w
S' = structure après traitement de w
p_w = nombre d'itérations de la boucle `TantQue` lors du traitement de w
μ = coût d'une itération de la boucle `TantQue`
ν = coût des trois instructions qui ne sont pas dans la boucle `TantQue` (`i=degre(w)`, `A[i]=w` et suppression de noeud).
Alors $c_w = \mu * p_w + \nu$. Le potentiel $\phi(S)$ est égal à (nombre de cases non vides de A) $* \lambda$.
Après traitement de w, chaque itération vidant 1 case, il y a $p_w - 1$ cases vides supplémentaires dans le tableau (puisqu'après la dernière itération l'affectation `A[i]=w` remplit une case).
Donc $\phi(S') - \phi(S) = -\lambda * (p_w - 1)$ et $\hat{c}_w = c_w + \phi(S') - \phi(S) = (\mu - \lambda)p_w + \lambda + \nu$.
En choisissant $\lambda = \mu$ on obtient $\hat{c}_w = \nu + \mu = \nu'$. D'où

$$c_2 \leq \sum_{w \text{ racine}} \hat{c}_w \leq \nu' * a(H)$$

Donc $c \leq \alpha_1 * D(n) + \nu' * a(H) + \alpha_3 D(n) \leq \alpha(a(H) + D(n))$, en posant $\alpha = \max(\nu', \alpha_1 + \alpha_3)$.

6. Phil A., à vérifier : me semble que dans la 6e ligne de l'algo il faudrait mettre à Nil chaque pointeur père des fils de z pour les mettre dans la liste des racines. Le coût de cette partie n'est alors pas constant comme affirmé dans la réponse suivante Mais comme il est inf ou égal à $D(n)$, ca ne change pas le raisonnement ensuite.

Procédure `ExtraireMinFBR`

```
ExtraireMinFBR(H)
  z:= min(H)
  Si fils(z) <> NIL Alors
      pere(fils(z)) = NIL
      Si z <> pred(z) Alors
          dans la liste des racines remplacer z par la liste de ses fils,
          en faisant les raccords necessaires
      Fin Si
      min(H) = fils(z) (* min(H) ne contient pas forcement la cle minimum *)
  Sinon
      Si z <> pred(z) Alors
          supprimer z de la liste des racines
          min(H) = pred(z) (* min(H) ne contient pas forcement la cle minimum *)
       Sinon
           min(H) = NIL
       Fin Si
   Fin Si
   Consolider(H)
   // le min a ete mis a jour : fait a la fin de Consolider
Fin ExtraireMinFBR
```

7. On extrait le minimum d'une file binomiale relâchée H de taille n et on produit la file H'.

Coût réel
Le coût réel de la première partie de la procédure, avant consolidation est une constante. Ensuite on consolide une file qui a au plus $a(H) + D(n) - 1$ racines (on a remonté au plus $D(n)$ fils de z et on a enlevé z de la liste des racines) : le coût réel de cette consolidation est inférieur ou égal à $\alpha(a(H) + D(n) - 1 + D(n))$. Le coût réel total est donc inférieur ou égal à $\gamma(a(H) + D(n))$ où γ est une constante correctement choisie.

Potentiel
On pose $\Phi(H) = \beta * a(H)$ (lorsque l'on considère une suite d'insertions et de suppressions, la file est initialement vide donc $\Phi(H_0) = 0$ et pour tout $i > 0$ on aura $\Phi(H_i) \geq 0 = \Phi(H_0)$).
Après l'étape de consolidation dans l'extraction du minimum, le nombre de racines dans H' est au plus $D(n) + 1$. Donc $\Phi(H') - \Phi(H) \leq \beta * (D(n) + 1) - \beta * a(H)$.

Coût amorti

$$\begin{aligned}
\hat{c} &\leq \gamma(a(H) + D(n)) + \beta * (D(n) + 1) - \beta * a(H) \\
&\leq (\gamma - \beta) * a(H) + \beta + (\gamma + \beta)D(n) \\
&\leq 2\gamma D(n) + \gamma \quad \text{(en choisissant } \beta = \gamma)
\end{aligned}$$

Le coût amorti est donc en $O(D(n)$, c'est-à-dire en $O(\lfloor \log(n) \rfloor)$.

Algorithmique Avancée

TD 3-4 : Arbres bicolores et tries

Exemplaire enseignant

Table des matières

1 Arbres bicolores

Solution

Conversion Les arbres bicolores permettent de représenter les arbres 2-3-4 avec des arbres binaires raisonnablement équilibrés et la conversion d'un type d'arbre à l'autre est aisée, comme on va le voir dans les exercices qui suivent.

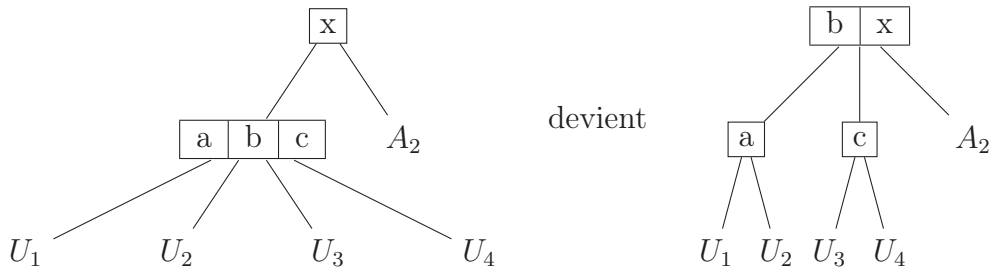
Exercice 1.1 ***: Arbres 2-3-4***

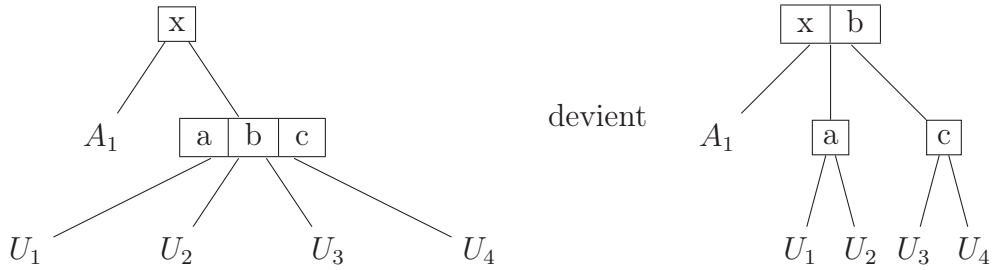
Question 1.1.1 Rappeler les règles d'éclatement d'un 4-nœud dans un arbre 2-3-4, lorsque les éclatements se font à la descente.

Question 1.1.2 Construire par adjonctions successives un arbre 2-3-4 contenant les clefs 8, 3, 2, 4, 1, 15, 10, 9, 11, 7, 6, 13, 12, 5, 14, 16, 17. On nommera cet arbre `A-ex1`.

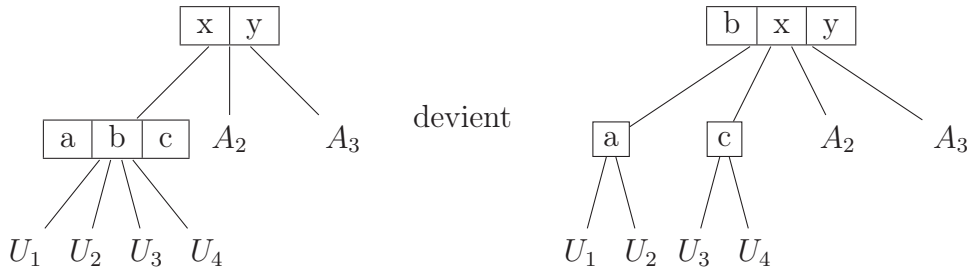
Solution

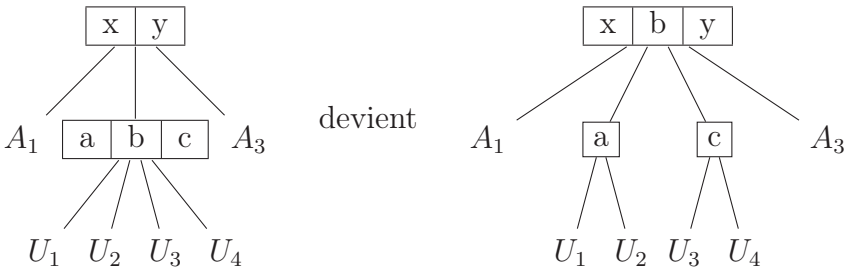
1. Le remplissage s'effectue par niveau descendant, chaque niveau étant complètement rempli avant de descendre. On est guidé dans les branches par l'ordre, ce qui ne laisse pas de choix. Du fait que les éclatements se font à la descente, le père d'un 4-noeud ne peut pas être un 4-noeud.
— si le père du 4-nœud est un 2-nœud, alors

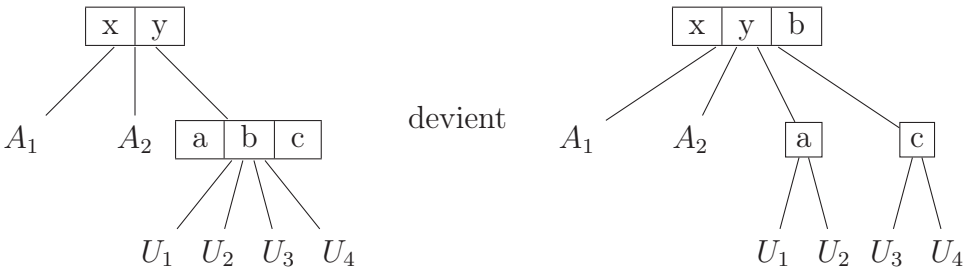




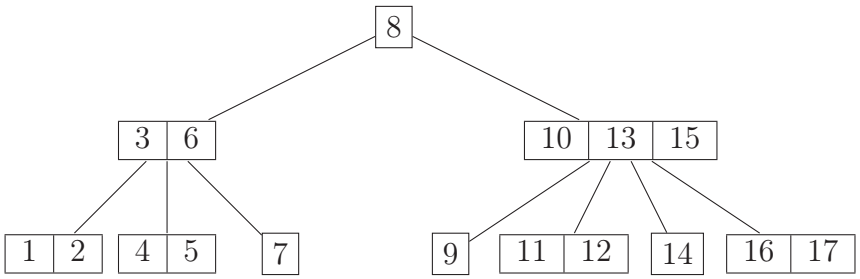
— si le père du 4-nœud est un 3-nœud, alors







2. Voilà l'arbre construit :



Définition des arbres bicolores

Un *arbre bicolore de recherche* est un arbre binaire de recherche complété dans lequel tout sommet possède une couleur (*blanc* ou *rouge*) et tout nœud interne possède une clef et qui vérifie :

(a) la racine est blanche
(b) les feuilles sont blanches (et ne possèdent pas de clef)
(c) le père d'un sommet rouge est blanc
(d) les chemins issus d'un même sommet et se terminant en une feuille ont le même nombre de sommets blancs.

Qualité

Les arbres bicolores sont des arbres raisonnablement équilibrés puisque la plus grande longueur de branche depuis la racine (cas pire : alternance 1 blanc, 1 rouge) ne dépasse pas deux fois la longueur de la plus petite branche (cas le meilleur : que des nœuds blancs). Et ceci simplement au prix de l'ajout d'un bit à chaque nœud pour coder la couleur. Ils permettent de représenter facilement un arbre 2-3-4 à l'aide d'un arbre binaire.

Grâce à cet équilibre relatif, le coût de l'algorithme de recherche dans un ABR est bien en $O(\log n)$. L"insertion et la suppression comportent évidemment une part de rééquilibrage.

Primitives

Pour manipuler les arbres bicolores, on a des primitives habituelles sur les arbres binaires :

`ArbreVide`	*Rien* $\rightarrow$ *arbre binaire*
`ArbreVide()`	*renvoie un arbre vide*
`ArbreBinaire`	*élément* $\times$ *arbre binaire* $\times$ *arbre binaire* $\rightarrow$ *arbre binaire*
`ArbreBinaire(x,G,D)`	*renvoie l'arbre binaire dont la racine a pour contenu l'élément x et dont les sous-arbres gauche et droit sont respectivement G et D*
`EstVide`	*arbre binaire* $\rightarrow$ *booléen*
`EstVide(T)`	*renvoie VRAI ssi T est un arbre binaire vide*
`Racine`	*arbre binaire* $\rightarrow$ *élément*
`Racine(T)`	*renvoie le contenu de la racine de l'arbre binaire T*
`SousArbreGauche`	*arbre binaire* $\rightarrow$ *arbre binaire*
`SousArbreGauche(T)`	*renvoie une copie du sous-arbre gauche de l'arbre binaire T*
`SousArbreDroit`	*arbre binaire* $\rightarrow$ *arbre binaire*
`SousArbreDroit(T)`	*renvoie une copie du sous-arbre droit de l'arbre binaire T*

et des primitives spécifiques aux arbres bicolores :

`Arbre`	*clef* $\times$ *couleur* $\times$ *arbre binaire* $\times$ *arbre binaire* $\rightarrow$ *arbre binaire*
`Arbre(a,c,G,D)`	*renvoie l'arbre binaire dont la racine a pour clef a et pour couleur c et dont les sous-arbres gauche et droit sont respectivement G et D*
`Clef`	*arbre binaire* $\rightarrow$ *clef*
`Clef(T)`	*renvoie la clef de la racine de T*
`Couleur`	*arbre binaire* $\rightarrow$ *couleur*
`Couleur(T)`	*renvoie la couleur de la racine de T*
`ModifierCouleur`	*arbre binaire* $\times$ *couleur* $\rightarrow$ *Rien*
`ModifierCouleur(T,c)`	*remplace par c la couleur de la racine de l'arbre binaire T*
`FeuilleBlanche`	*Rien* $\rightarrow$ *arbre binaire*
`FeuilleBlanche()`	*renvoie l'arbre binaire réduit à une feuille de couleur blanche sans clef*
`EstFeuilleBlanche`	*arbre binaire* $\rightarrow$ *booléen*
`EstFeuilleBlanche(T)`	*renvoie VRAI ssi T est un arbre bicolore réduit à une feuille blanche sans clef*

Exercice* 1.2 : *Transformation d'un arbre 2-3-4 en arbre bicolore

Principe : Pour obtenir un arbre bicolore à partir d'un arbre 2-3-4, on procède de la façon suivante :

- l'arbre vide est transformé en une feuille blanche (sans clef)
- chaque 2-nœud prend la couleur blanche et ses deux sous-arbres sont transformés en arbres bicolores
- un 3-nœud, qui compte deux clefs $a < b$, se scinde en deux nœuds de l'arbre bicolore ; ces deux nœuds contiennent respectivement les clefs a et b. Il y a deux possibilités : ou bien b est à la racine du sous-arbre droit de a ou bien a est à la racine du sous-arbre gauche de b. Le fils prend la couleur rouge et le père prend la couleur blanche. Les trois sous-arbres du 3-nœud sont eux-mêmes transformés en arbres bicolores et deviennent les sous-arbres des nœuds contenant les clefs a et b
- un 4-nœud, qui compte trois clefs $a < b < c$, se scinde en trois nœuds de l'arbre bicolore ; ces trois nœuds contiennent respectivement les clefs a, b et c. La clef a est à la racine du sous-arbre gauche de b et la clef c est à la racine du sous-arbre droit de b. Les fils prennent la couleur rouge et le père prend la couleur blanche. Les quatre sous-arbres du 4-nœud sont eux-mêmes transformés en arbres bicolores et deviennent les sous-arbres des nœuds contenant les clefs a et c.

Question 1.2.1 Illustrer le principe énoncé ci-dessus (au moyen de petits dessins).

Question 1.2.2 Construire un arbre bicolore, que l'on nommera `B-ex1`, transformé de l'arbre 2-3-4 `A-ex1`.

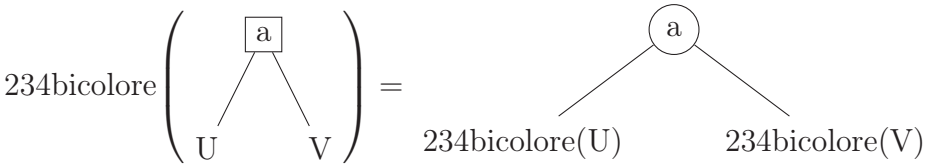
Question 1.2.3 Écrire l'algorithme de transformation d'un arbre 2-3-4 en arbre bicolore. On dispose des primitives des arbres 2-3-4 du cours.

Question 1.2.4 Montrer que l'arbre ainsi obtenu est un arbre bicolore et encadrer la hauteur de cet arbre.

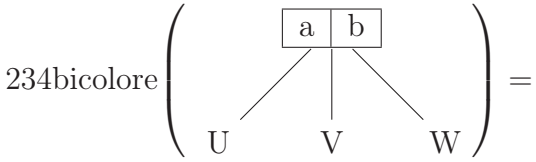
Solution

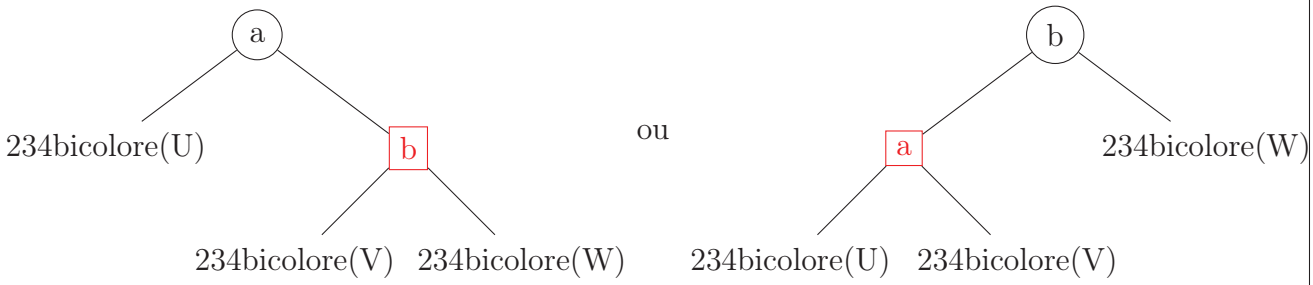
1. Dans mes dessins, les nœuds blancs seront ronds et noirs, et les nœuds rouges seront carrés.

Règle 2 :

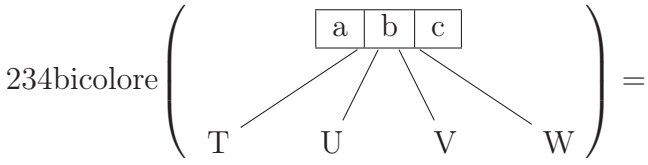


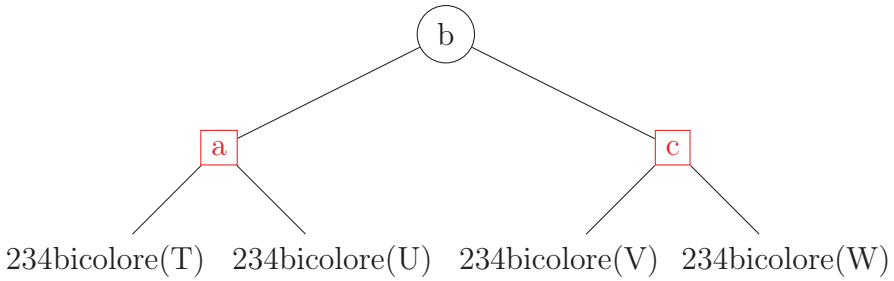
Règle 3 :



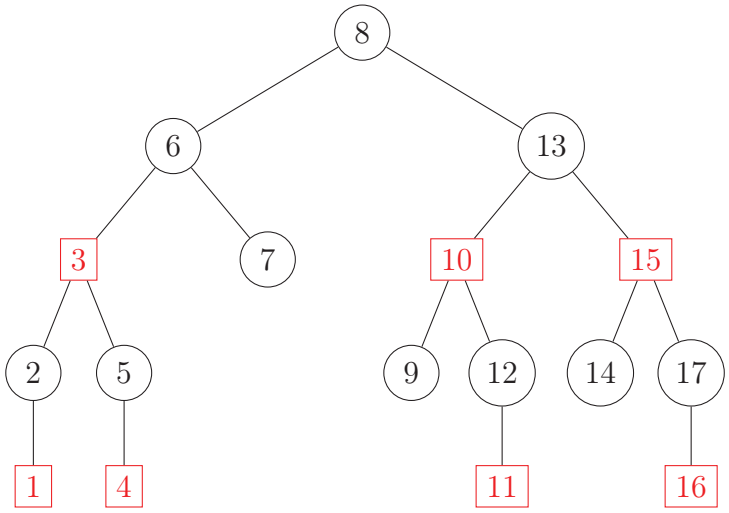


Règle 4 :





2.



3.

```
234bicolore (A)
si EstVide(A) alors retourne (FeuilleBlanche())
si Degre(A)=2 alors
  retourne (Arbre (Elem-1(A), blanc, 234bicolore(SSab-1(A)), 234bicolore(SSab-2(A))))
si Degre(A)=3 alors
  retourne (Arbre (Elem-2(A), blanc,
            Arbre (Elem-1(A), rouge, 234bicolore(SSab-1(A)), 234bicolore(SSab-2(A))))
            234bicolore(SSab-3(A)))
si Degre(A)=4 alors
  retourne (Arbre (Elem-2 (A), blanc,
            Arbre (Elem-1(A), rouge, 234bicolore(SSab-1(A)), 234bicolore(SSab-2(A))))
            Arbre (Elem-3(A), rouge, 234bicolore(SSab-3(A)), 234bicolore(SSab-4(A))))
Fin 234bicolore
```

4. (a) Les feuilles sont introduites par l'algorithme au niveau des arbres vides sous la forme de feuilles blanches sans clef donc toutes les feuilles de l'arbre transformé sont blanches et sans clef

(b) Par construction, on introduit un sommet rouge soit comme le fils rouge d'un père blanc dans le cas d'un 3-neoud, soit comme l'un des deux fils rouges d'un père blanc dans le cas d'un 4-nœud, donc tous les sommets rouges ont un père blanc.

(c) Par récurrence sur la hauteur de l'arbre 2-3-4.

Hypothèse de récurrence : pour tout arbre 2-3-4 de hauteur h la transformation décrite produit un arbre bicolore pour lequel tout chemin dans l'arbre bicolore d'un nœud n, de hauteur ℓ dans l'arbre 2-3-4, à une feuille contient exactement ℓ nœuds blancs hormis le nœud lui-même.

Si la hauteur de l'arbre 2-3-4 est 0, alors l'arbre est limité à un nœud, tout chemin d'un nœud de l'arbre à une feuille contient exactement 0 nœud blanc hormis lui-même soit la hauteur de ce nœud, donc la propriété est vraie pour 0.

Supposons que tout arbre résultant de la transformation d'un arbre 2-3-4 de hauteur h vérifie la propriété précédente. Soit A un arbre 2-3-4 de hauteur $h+1$. Tous les sous-arbres de A sont des arbres 2-3-4 de hauteur h exactement puisque toutes les feuilles d'un arbre 2-3-4 sont au même niveau. Chacun de leur transformés est donc un arbre bicolore qui vérifie l'hypothèse de récurrence.

Soit n un nœud de l'arbre :
— ou bien ce nœud était l'étiquette d'un nœud de hauteur $\ell \leq h$ dans l'un des fils de l'arbre 2-3-4 et l'hypothèse de récurrence s'applique à ce fils pour nous donner le résultat.
— ou bien c'était une des étiquettes de la racine de l'arbre 2-3-4. Un chemin qui va de cette étiquette à une feuille dans l'arbre transformé est un chemin qui passe éventuellement par une autre étiquette qui sera coloriée en rouge puis par l'étiquette du fils d'un des sous-arbres de ce nœud. Par hypothèse de récurrence, le chemin qui va de ce nœud à la feuille considérée a exactement h nœuds blancs et lui-même est blanc, qu'on ait appliqué la règle de transformation 2, 3 ou 4. Donc le chemin d'une étiquette de la racine à une feuille passe par $h+1$ nœuds blancs hormis éventuellement cette étiquette.

donc la propriété est vraie pour un arbre 2-3-4 de hauteur $h+1$ et la propriété est finalement vraie pour un arbre 2-3-4 quelle que soit sa hauteur : le transformé d'un arbre 2-3-4 par l'algorithme de l'énoncé produit un arbre bicolore qui contient exactement les mêmes nœuds.

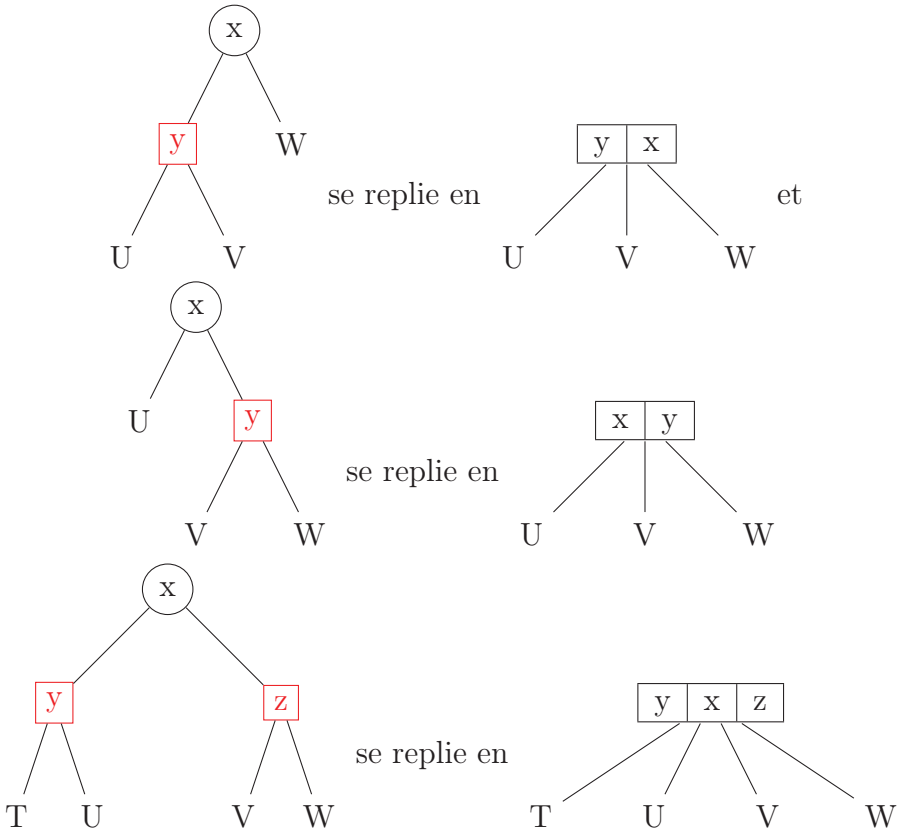
Exercice* 1.3 *: Transformation d'un arbre bicolore en arbre 2-3-4

Question 1.3.1 Donner le principe de la transformation d'un arbre bicolore en arbre 2-3-4. Illustrer ce principe au moyen d'un exemple.

Question 1.3.2 Écrire l'algorithme de transformation d'un arbre bicolore en arbre 2-3-4. La création d'arbres 2-3-4 sera assurée par des primitives `ArbreVide234`, et `Arbre234` qui prend en argument une liste de clés et une liste d'arbres 2-3-4.

Solution

1. L'arête d'un nœud rouge à son père correspond à un pli de remontée : le nœud rouge va prendre place avec son père dans un 3- ou 4-nœud.
Par exemple si on replie ainsi `B-ex1`, on obtient à nouveau `A-ex1`.



2. On aurait aussi pu avoir trois constructeurs pour chacun des types de nœuds des arbres 2-3-4, ce qui aurait été plus précis.

```
bicolore234 (A)
  si EstFeuilleBlanche (A) alors retourne (ArbreVide234())
  LR <- [Clef (A)]
  LF <- []
  G <- SousArbreGauche(A)
  D <- SousArbreDroit(A)
  si non (EstFeuilleBlanche (G))
     alors si Couleur (G)=rouge
           alors LR <- Clef (G) + LR
                 LF <- [bicolore234(SousArbreGauche(G)),
                              bicolore234(SousArbreDroit(G))] + LF
           sinon LF <- [bicolore234(G)] + LF
  si non (EstFeuilleBlanche (D))
     alors si Couleur (D)=rouge
           alors LR <- LR + Clef (SousArbreDroit(A))
                 LF <- LF + [bicolore234(SousArbreGauche(D)),
                                    bicolore234(SousArbreDroit(D))]
           sinon LF <- LF + [bicolore234(G)]
  retourne (Arbre234 (LR, LF))
```

Exercice 1.4 ***: Rotations dans un arbre binaire***

On définit les rotations simples `RotationGauche` et `RotationDroite` de spécifications :

`RotationGauche`	*arbre binaire* $\rightarrow$ *arbre binaire*
`RotationGauche(T)`	*renvoie l'arbre obtenu en faisant basculer T vers la gauche*
`RotationDroite`	*arbre binaire* $\rightarrow$ *arbre binaire*
`RotationDroite(T)`	*renvoie l'arbre obtenu en faisant basculer T vers la droite*

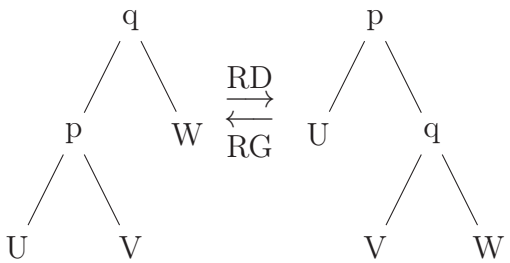
ainsi que les rotations doubles `RotationGaucheDroite` et `RotationDroiteGauche` de spécifications :

`RotationGaucheDroite :`	*arbre binaire* $\rightarrow$ *arbre binaire*
`RotationGaucheDroite(T)`	*renvoie l'arbre obtenu en faisant basculer d'abord le sous-arbre gauche de T vers la gauche puis l'arbre T ainsi modifié vers la droite*
`RotationDroiteGauche :`	*arbre binaire* $\rightarrow$ *arbre binaire*
`RotationDroiteGauche(T)`	*renvoie l'arbre obtenu en faisant basculer d'abord le sous-arbre droit de T vers la droite puis l'arbre T ainsi modifié vers la gauche*

Question 1.4.1 Écrire la définition de l'une des deux rotations simples et la définition de l'une des deux rotations doubles.

Solution

1. On veut un rappel du cours, un dessin ou le code ici ? Par précaution, voici les trois :
— rotation droite : $RD(< q, < p, U, V >, W >) =< p, U, < q, V, W >>$.
— rotation gauche : $RG(< p, U, < q, V, W >>) =< q, < p, U, V >, W >$.

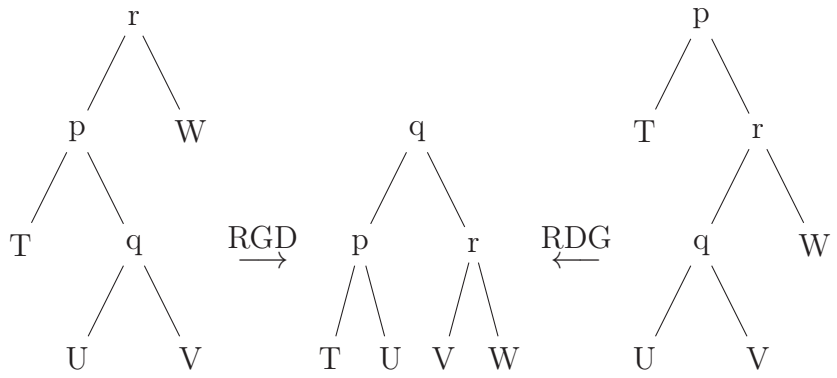



```
RD(A)
  si EstVide(A) ou EstVide(SousArbreGauche(A))
  alors retourne (A)
  sinon p <- Racine(SousArbreGauche(A))
        q <- Racine(A)
```

```
        U <- SousArbreGauche(SousArbreGauche(A))
        V <- SousArbreDroit(SousArbreGauche(A))
        W <- SousArbreDroit(A)
        retourne (ArbreBinaire(p,U,ArbreBinaire(q,U,V),W))
  fin si
fin RD
```

— rotation gauche droite : $RGD(< r, < p, T, < q, U, V >>, W >) =< q, < p, T, U >, < r, V, W >>$.

— rotation droite gauche : $RDG(< p, T, < r, < q, U, V >, W >>) =< q, < p, T, U >, < r, V, W >>$.




```
RGD(A)
  si EstVide(A) ou EstVide(SousArbreGauche(A))
      ou EstVide{SousArbreDroit(SousArbreDroit(A))}
  alors retourne (A)
  sinon
    p <- Racine(SousArbreGauche(A))
    q <- Racine(SousArbreDroit(SousArbreGauche(A)))
    r <- Racine(A)
    T <- SousArbreGauche(SousArbreGauche(A))
    U <- SousArbreGauche(SousArbreDroit(SousArbreGauche(A)))
    V <- SousArbreDroit(SousArbreDroit(SousArbreGauche(A)))
    W <- SousArbreDroit(A)
    retourne (ArbreBinaire(q,ArbreBinaire(p,T,U),ArbreBinaire(r,V,W)))
  fin si
fin RGD
```

Exercice 1.5 ***: Insertion dans un arbre bicolore***

Principe : L'insertion dans un arbre bicolore suit le principe de l'insertion dans un arbre 2-3-4. Nous travaillerons ici sur l'insertion avec éclatements à la descente.

Question 1.5.1 Transposer dans les arbres bicolores les différents cas d'insertion d'une clef dans une feuille d'un arbre 2-3-4.

Question 1.5.2 Transposer dans les arbres bicolores les différents cas d'éclatement d'un 4-nœud.

Question 1.5.3 Réaliser l'insertion des clefs 13.1, 13.3, 12.5, 12.8 dans l'arbre bicolore `B-ex1`.

Question 1.5.4 Écrire l'algorithme d'insertion d'une clef dans un arbre bicolore.

Question 1.5.5 Ci-dessous figure un algorithme d'insertion extrait du livre "Introduction à l'algorithmique" de Cormen, Leiserson, Rivest et Stein. Est-ce le même que le nôtre ?

Algorithmes d'insertion dans un ABR et dans un arbre bicolore

```
ARBRE-INSERER(T,z)
  y <- NIL
  x <- racine[T]
  tantque x <> NIL
    faire y <- x
          si cle[z] < cle[x]
             alors x <- gauche[x]
             sinon x <- droit[x]
  p[z] <- y
  si y = NIL
     alors racine[T] <- z
     sinon si cle[z] < cle[y]
              alors gauche[y] <- z
              sinon droit[y] <- z
```

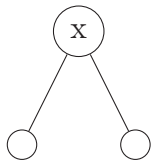
```
ROTATION-GAUCHE(T,x)
  y <- droit[x]
  droit[x] <- gauche[y]
  si gauche[y] <> NIL
     alors p[gauche[y]] <- x
  p[y] <- p[x]
  si p[x] = NIL
     alors racine[T] <- y
     sinon si x = gauche[p[x]]
              alors gauche[p[x]] <- y
              sinon droit[p[x]] <- y
  gauche[y] <- x
  p[x] <- y
```

Le code de ROTATION-DROITE est similaire au code de ROTATION-GAUCHE.

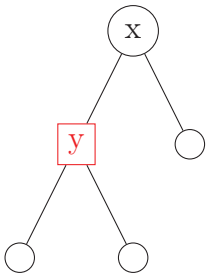
```
RN-INSERER(T,x)
  ARBRE-INSERER(T,x)
  couleur[x] <- ROUGE
  tantque x <> racine[T] et couleur[p[x]] = ROUGE
    faire si p[x] = gauche[p[p[x]]]
            alors y <- droit[p[p[x]]]
                 si couleur[y] = ROUGE
                    alors couleur[p[x]] <- NOIR
                          couleur[y] <- NOIR
                          couleur[p[p[x]]] <- ROUGE
                          x <- [p[x]]
                    sinon si x = droit[p[x]]
                             alors x <- p[x]
                                   ROTATION-GAUCHE(T,x)
                          couleur[p[x]] <- NOIR
                          couleur[p[p[x]]] <- ROUGE
                          ROTATION-DROITE(T,p[p[x]])
                    sinon (comme la clause alors
                             en echangeant droit et gauche)
  couleur[racine[T]] <- NOIR
```

Solution

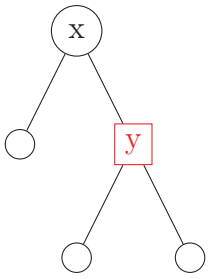
1.



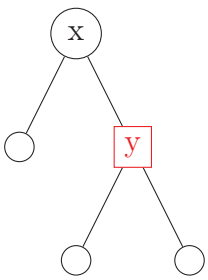
devient après insertion



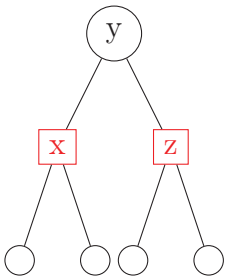
ou (suivant la valeur de y)



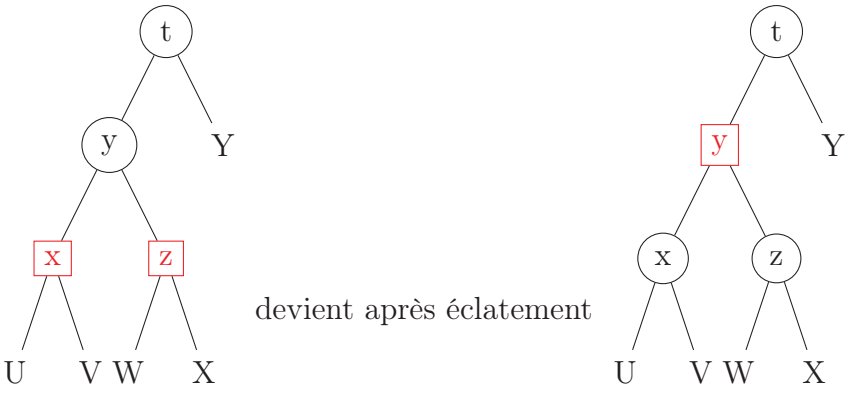
Tous les cas du 3-nœud ne sont pas traités :



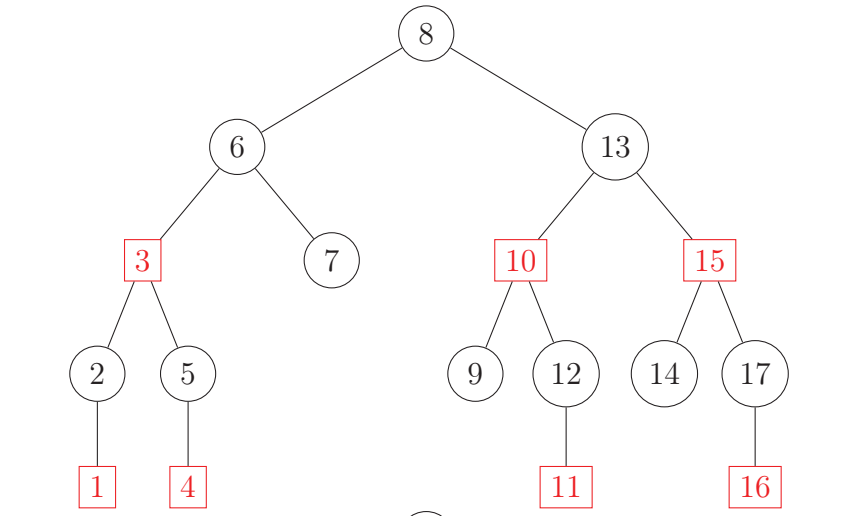
devient après rotation RG et insertion ($z > y > x$)

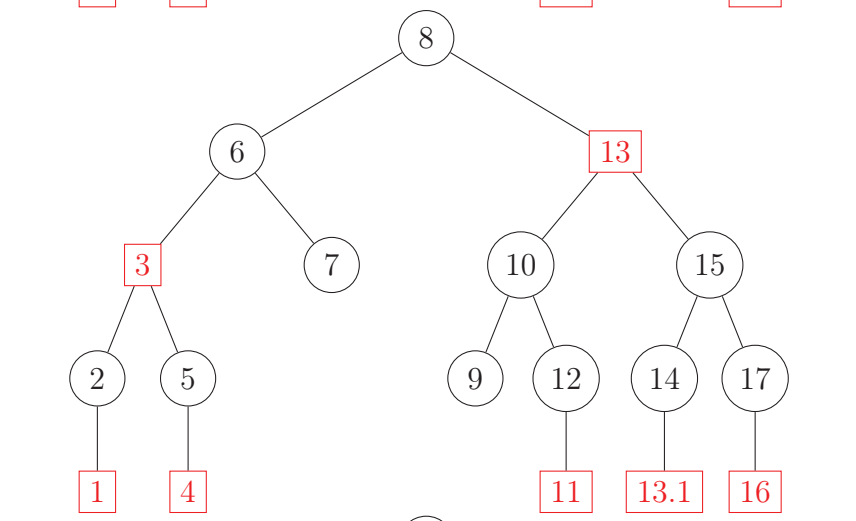


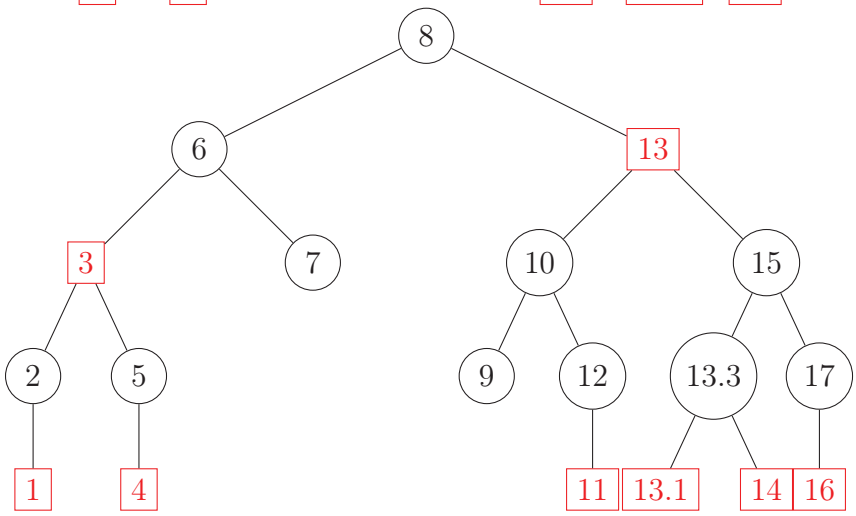
2. Tous les cas d'éclatement ne sont pas traités :

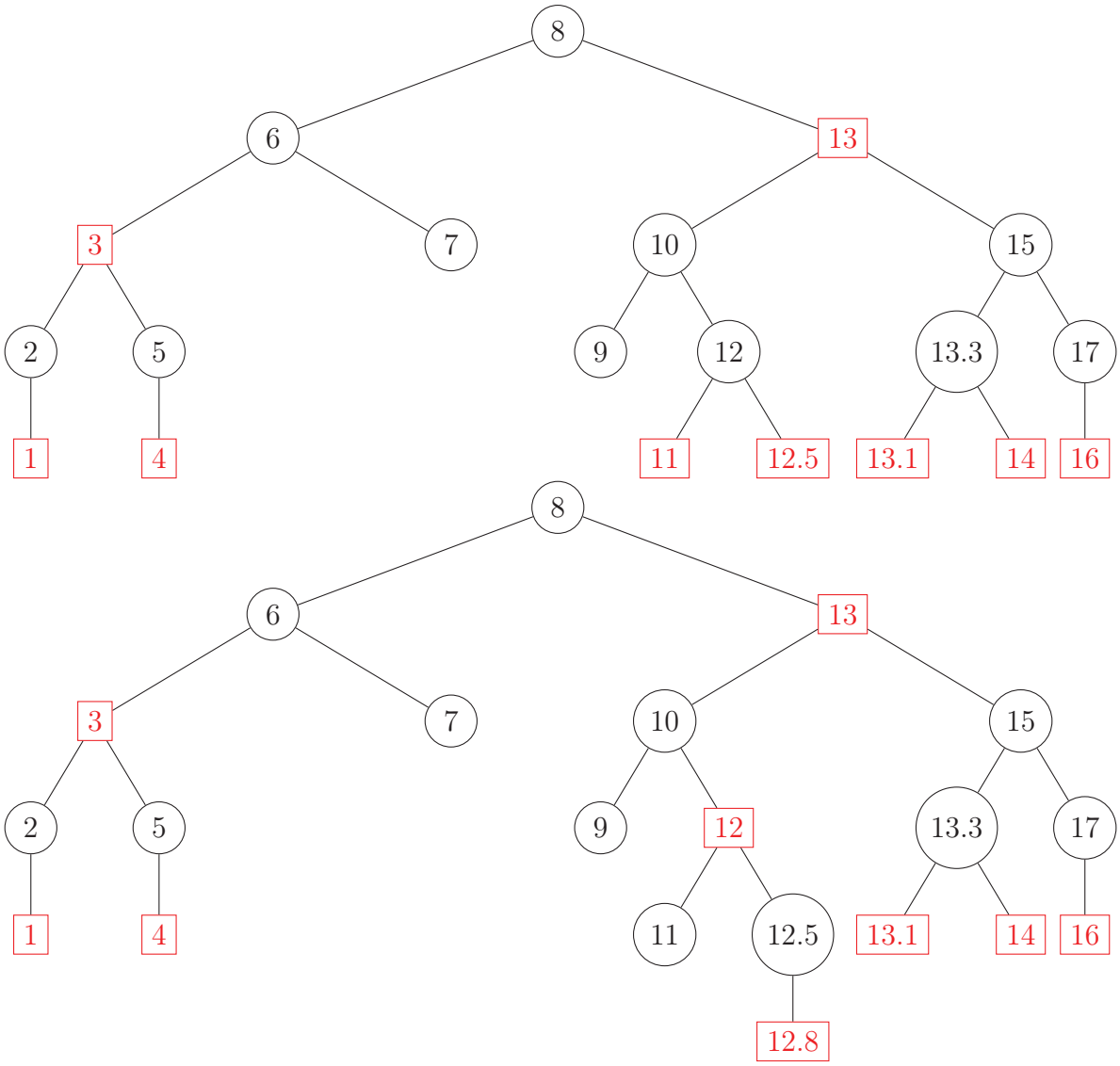

t
y
Y
x
z
U
V
W
X
devient après éclatement
t
y
Y
x
z
U
V
W
X

3.


8
6
13
3
7
10
15
2
5
9
12
14
17
1
4
11
16


8
6
13
3
7
10
15
2
5
9
12
14
17
1
4
11
13.1
16


8
6
13
3
7
10
15
2
5
9
12
13.3
17
1
4
11
13.1
14
16



VERIFIER algo : cf mail Elie du 19/10/2017

4. Il faut un programme principal pour éventuellement remettre la racine à blanc.

```
BicInsertion(A, v)

  si EstVide(A) alors
    B = ArbreBic(v, blanc, ArbreBicVide(), ArbreBicVide())
  sinon
    B = BicIns(A,v)

  renvoyer ArbreBic( clef(B), blanc, SousArbreGauche(B), SousArbreDroit(B))

fin BicInsertion
```

La fonction d'insertion avec éclatement à la descente

```
BicIns(A, v)

  G = SousArbreGauche(A)
  D = SousArbreDroit(A)
  si couleur(G) = rouge et couleur(D) = rouge alors
    B = ArbreBic(clef(A), rouge, ArbreBic(clef(G), blanc, SousArbreGauche(G), SousArbreDroit(G)),
                           ArbreBic(clef(D), blanc, SousArbreGauche(D), SousArbreDroit(D)))
  sinon
    B = A

  G = SousArbreGauche(B)
  D = SousArbreDroit(B)

  si v < clef(B) alors
    si EstVide(G) alors
      G = ArbreBic(v, rouge, ArbreBicVide(), ArbreBicVide())
      renvoyer ArbreBic(clef(B), blanc, G, D)
```

```
      sinon si EstVide(D) alors
        si v < clef(G) alors
          C = ArbreBic(v, rouge, ArbreVide(), ArbreVide())
          B = ArbreBic(clef(B), rouge, ArbreBic(clef(G), blanc, C, ArbreVide()), D)
            rem : les couleurs ne sont pas correctes ici, mais apres la rotation, ce sera OK
          renvoyer RD(B)
        sinon
          C = ArbreBic(v, blanc, ArbreVide(), ArbreVide())
          B = ArbreBic(clef(B), rouge, ArbreBic(clef(G), rouge, ArbreVide(), C), D)
            rem : les couleurs ne sont pas correctes ici, mais apres la rotation, ce sera OK
          renvoyer RGD(B)

      sinon
        renvoyer BicIns(G, v)

    sinon (rem : v > clef(B))
      rem : cas symetrique au precedent

  fin BicIns
```

5. Éclatements à la remontée au lieu de la descente...

2 Arbres de Recherche versus Tries

Exercice 2.1 ***: Comparaisons sur des exemples***

Question 2.1.1 Construire l'arbre binaire de recherche (ABR) obtenu par l'insertion successive des lettres : A, S, E, R, C, H, I, N, G, X, M, P et L.
On utilise l'ordre alphabétique pour comparer deux lettres.

Question 2.1.2 Que se passe-t-il si on construit l'ABR avec la succession suivante de lettres : L, S, A, R, E, H, C, N, I, X, G, P et M ?

Question 2.1.3 En utilisant le codage ci-dessous, et les deux exemples précédents, parmi les modèles d'ABR, d'arbre digital et d'arbre lexicographique lesquels sont sensibles à l'ordre d'insertion des lettres ?

A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001

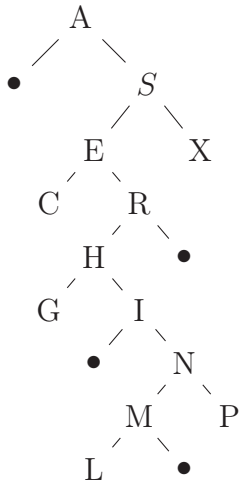
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100

Remarque : on lit le bit de poids fort (le plus à gauche) d'abord.

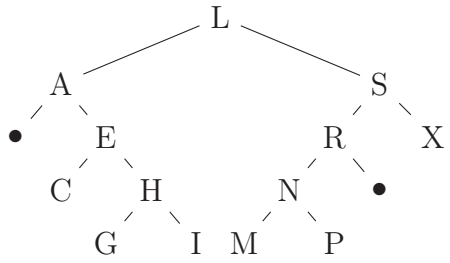
Question 2.1.4 Construire les arbres binaire de recherche, digital et lexicographique avec la succession de lettres : N, A, S, P, I, X.

Solution

1.

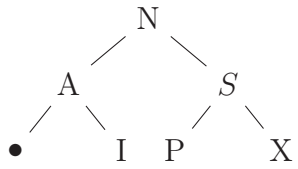


2.

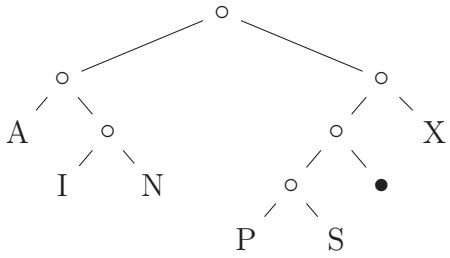


3. L'ordre d'insertion est important dans l'ABR et l'arbre digital. Il n'importe pas dans l'arbre lexicographique.

4. Pour l'ABR et l'arbre digital on trouve le même arbre.



Pour l'arbre lexicographique, ou trie binaire, on obtient :



Exercice 2.2 ***: R-trie ou arbres de la Briandais ?***

On se place sur l'alphabet à 4 lettres A,C,G et T encodant les séquences d'ADN. On rappelle qu'un nœud d'un R-trie contient une valeur non vide lorsqu'il représente une clé. Cela permet notamment d'encoder deux mots dont l'un est préfixe de l'autre. On représente le 4-trie vide par $\emptyset$.

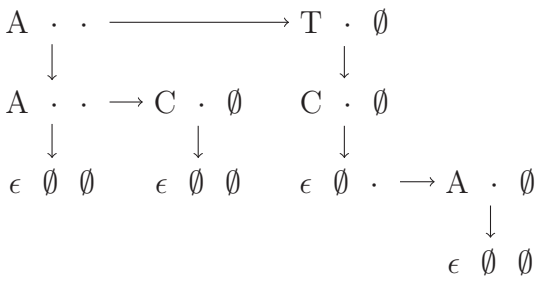
Question 2.2.1 Encoder les 4 mots AA, AC, TCA et TC dans un 4-trie (avec tous les liens, mêmes ceux étant vides).

Question 2.2.2 Dessiner le 4-trie (avec tous les liens, mêmes ceux étant vides) construits sur les mots suivants : TACG ; AAT ; AT ; CGGA et TAC.

On se rend compte que cet arbre contient beaucoup de pointeurs vers le trie vide, et beaucoup de

nœuds internes vides (aucun mot ne se termine en ce nœud). L'idée des arbres de la Briandais, pour représenter les R-tries, consiste à remplacer le tableau de taille R de chaque nœud par une liste **triée** ne contenant que les lettres utiles. On a besoin d'un nouveau caractère (ϵ) pour indiquer la fin d'un mot. On suppose que les frères sont ordonnés selon l'ordre alphabétique : $\epsilon < A < C < G < T$. Le pointeur vers `Nil` est représenté par $\emptyset$.

Ainsi, les mots AA, AC, TCA et TC sont encodés en arbre de la Briandais ainsi :



Question 2.2.3 Encoder l'ensemble des mots de la question 2 dans un arbre de la Briandais.

Question 2.2.4 Donner les spécifications des primitives afin de construire un arbre de la Briandais (par succession d'ajouts de mots), d'y faire la recherche d'un mot et la suppression d'un mot.

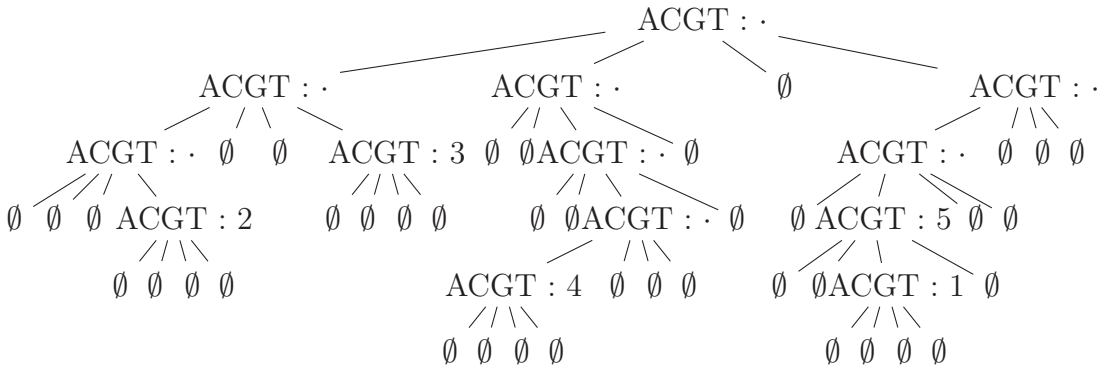
Question 2.2.5 Donner le pseudo-code de l'insertion d'un mot dans un arbre de la Briandais.

Question 2.2.6 Donner le pseudo-code de la suppression d'un mot dans un arbre de la Briandais.

Solution

1. Il faut dessiner un peu...

2.



3. Il faut dessiner un peu...

4. On a besoin de
- BRD : caractère, BRDarbre, BRDarbre → BRDarbre : renvoie le BRDarbre correspondant
- pour un BRDarbre A, on a les les fonctions clef(A), fils(A), frere(A) qui renvoient resp. la clef, et des BDR.
- Arbre vide : $\emptyset$ ou Nil

5. Rappel : les lettres "frères" sont triées dans l'ordre alphabétique (avec ϵ le premier caractère). Les fonctions `tete` et `queue` renvoient respectivement le premier caractères et tout le mot privé du premier caractère d'une chaîne de caractères.
On commence par définir une fonction construisant un BRD à partir d'un mot.

```
BRDcons(m)

  si m = '' alors
    renvoyer BRD(epsilon, Nil, Nil)
```

```
  sinon
    renvoyer BRD(tete(m), BRDcons(queue(m)), Nil)

fin BRDcons
```

Pour la fonction d'insertion, on suppose que le mot à insérer ne figure pas encore dans l'arbre.

```
BRDinsertion(A,m)

  si m = '' alors
    renvoyer BRD(epsilon, Nil, A)
  si A = Nil alors
    renvoyer BRDcons(m)

  t = tete(m)
  si clef(A) < t alors
    Fr = BRDinsertion(frere(A), m)
    renvoyer BRD(clef(A), fils(A), Fr)

  si clef(A) = t alors
    Fi = BRDinsertion(fils(A), queue(m))
    renvoyer BRD(clef(A), Fi, frere(A))

  si clef(A) > t alors
    renvoyer BRD(t, BRDcons(queue(m)), A)

fin BRDinsertion
```

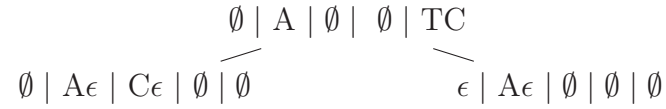
6. On peut faire appel à une nouvelle primitive : BRDnbMots : BRDarbre → entier : renvoie le nombre de mots encodés dans l'arbre.
On ne donnera pas le code de cette fonction, qui sera étudié en devoir de programmation. Mais il suffit d'ajouter un champ aux nœuds qui est incrémenté à chaque insertion de mot.
On suppose que le mot à supprimer apparaît dans l'arbre.

```
BRDsuppression(A, m)

  si m = '' alors            #alors c'est que la racine de A contient epsilon
    renvoyer frere(A)

  si BRDnbMots(A) = 1 alors
    renvoyer Nil

  t = tete(m)
  si clef(A) < t alors
    renvoyer BRD(clef(A), fils(A), BRDsuppression(frere(A), m))
  sinon                          #alors clef(A) = t  et l'arbre encode plusieurs mots
    renvoyer BRD(clef(A), BRDsuppression(fils(A), queue(m)), frere(A))

fin BRDsuppression
```

Exercice 2.3 ***: PATRICIA Trie***

Le but des arbres PATRICIA (Practical Algorithm To Retrieve Information Coded In Alphanumeric) est de réduire la taille des R-tries tout en conservant une recherche efficace. Pour ce faire, plutôt que chaque nœud interne permette de distinguer une lettre, il permet de distinguer la plus longue sous-chaîne de lettres commune à plusieurs mots.

Les 4 mots AA, AC, TC et TCA sont représentés par l'arbre PATRICIA suivant :



On remarque que le premier caractère permet toujours de distinguer les sous-chaînes stockées dans chaque nœud interne.

Question 2.3.1 Représenter l'arbre PATRICIA des mots : TACG ; AAT ; AT ; CGGA et TAC. Le comparer à l'arbre de la Briandais de l'exercice précédent.

Question 2.3.2 Donner les spécifications des primitives afin de construire un arbre PATRICIA (par succession d'ajouts de mots), d'y faire la recherche d'un mot et la suppression d'un mot.

Question 2.3.3 Donner le pseudo-code de l'insertion d'un mot dans un arbre PATRICIA.

Question 2.3.4 Donner le pseudo-code de la suppression d'un mot dans un arbre PATRICIA.

Question 2.3.5 Donner le pseudo-code de la fusion de deux arbres PATRICIA en un seul.

Solution

1.

2. On a besoin de
— PATvide : → PATarbre : renvoie un noeud vide avec tous les fils à Nil

3. PATcons prend un mot et construit l'arbre correspondant à ce mot (un seul noeud). Attention, PATcons, ne rajoute pas le caractère epsilon !

```
PATcons(m)

  A = PATvide()
  A.cle(tete(m)) = m

fin PATinsertion
```

On suppose que A ne contient pas m

```
PATinsertion(A, m)

  t = tete(m)
  c = A.cle(t)
  si c = empty alors
    A.cle(t) = concat(m, 'epsilon')
    renvoyer A

  si estPrefixe(c, m) alors       # c est-il prefixe de m ? eventuellement c=m
    A.fils(t) = PATinsertion(A.fils(t), m[lg(c)+1 .. lg(m)])
    renvoyer A

  p = prefixe(c, m)        #le plus long prefixe commun
  F = PATcons(c[lg(p)+1 .. lg(c)])
  F.fils(c[lg(p)+1]) = A.fils(tete(c))
  n = m[lg(p)+1 .. lg(m)]
  F.cle(tete(n)) = concat(n, 'epsilon')
  A.cle(t) = p
  A.fils(t) = F
  renvoyer A

fin PATinsertion
```

4. On suppose que m est dans A

```
PATsuppression(A, m)

  t =tete(m)
  si m = '' alors
    A.cle(epsilon) = Nil
    renvoyer A

  si A.cle(t) = concat(m, 'epsilon') alors
    A.cle(t) = Nil
    renvoyer A

  p = prefixe(A.cle(t), m)
  A.fils(t) = PATsuppression(A.fils(t), m[lg(p)+1 .. lg(m)])
  s = []
  pour t dans l'alphabet
    s = s + [A.fils(t)]
  si len(s) = 1 alors
    A.cle(tete(s[1])) = concat(A.cle(tete(s[1])), s[1])
    A.fils(tete(s[1])) = Nil

  renvoyer A

fin PATsuppression
```

5. On fusionne A et B. Que se passe-t-il s'il y a des mots en commun ?

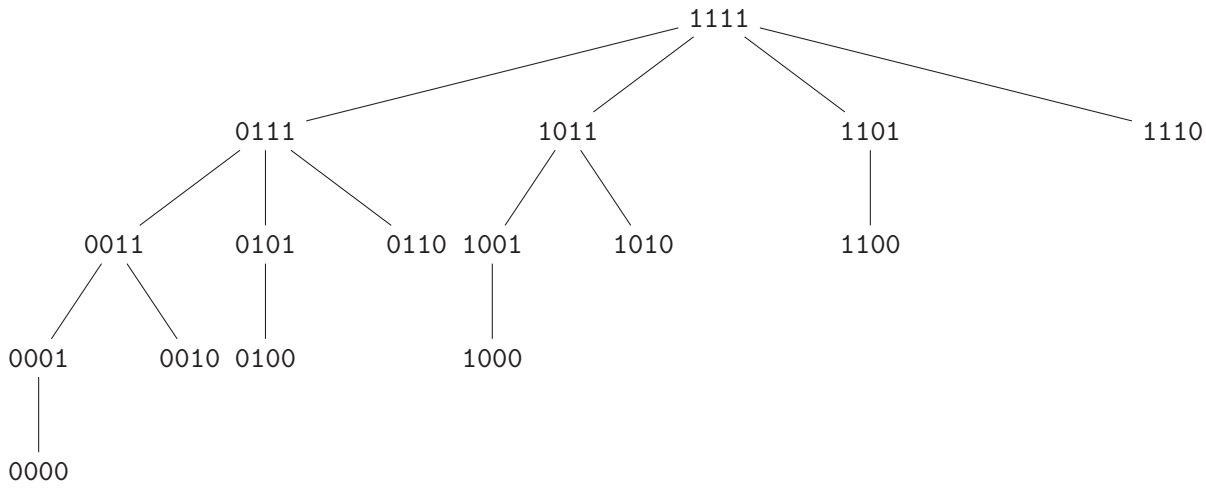
```
PATfusion(A, B)

  pour a in alphabet
    si A.cle(a) = empty alors
      A.cle(a) = B.cle(a)
      A.fils(a) = B.fils(a)
      renvoyer A

    si A.cle(a) = B.cle(a) alors
      A.fils(a) = PATfusion(A.fils(a), B.fils(a))
      renvoyer A
```

```
        p = prefixe(A.cle(a), B.cle(a))
        c = A.cle(a)
        m = B.cle(a)
        F = PATcons(c[lg(p)+1 .. lg(c)])
        F.fils(c[lg(p)+1]) = A.fils(a)
        G = PATcons(m[lg(p)+1 .. lg(m)])
        G.fils(m[lg(p)+1]) = B.fils(a)
        A.cle(a) = p
        A.fils(a) = PATfusion(F, G)
        renvoyer A

    fin PATsuppression
```

1 Partiel de novembre 2004

1.1 Arbres binômiaux

Question 1



Question 2
Procédons par récurrence :
Soit P_n la propriété :

Dans l'arbre B_k les nœuds à profondeur i ont $k-i$ 1

`Initialisation :`
La propriété P_0 est vraie.
Supposons la propriété P_k vraie quel que soit $k \leq n$ et montrons P_{n+1}.
Considérons l'arbre B_{n+1}, soit x un nœud de B_{n+1} à profondeur i. Par construction des arbres binômiaux le fils le plus à gauche de la racine de B_{n+1} est l'arbre B_n où on a prefixé d'un 0 les clefs de B_n. Toujours par constructiondes arbres binômiaux, l'arbre obtenu en supprimant le fils gauche de la racine de B_{n+1} est l'arbre B_n où on aura prefixé par un 1 les clefs de B_n.
Si x est un nœud du fils gauche de la racine de B_{n+1} alors par hypothèse de récurrence x a $i-1$ 0 dans sa clef, plus un pour le prefixe i.e. $n+1-i$ 1. Sinon, on a : les nœuds à profondeur i ont exactement $n-i$ bits à 1 parmi leur n derniers bits et 1 bit de préfixe, i.e. $n+1-i$ bits à 1 au total ; P_{n+1} est donc vraie.
La propriété P_n étant vraie au rang 0 et étant héréditaire elle est vraie à tout rang $n \in mathbbN$.
Question 3
Par la question précédente, les nœuds ayant exactement $n-i$ 1 dans leur clef sont les nœuds à profondeur i de B_n. Le nombre de nœuds à profondeur i d'un arbre binômial est C_n^i.
Question 4
Montrons cette propriété par récurrence sur n. Soit P_n la propriété :

Dans l'arbre B_n le degré d'un nœud est égal au nombre de 1 *à droite du* 0 *le plus à droite de son étiquette (et si l'étiquette ne contient pas de* 0*, le degré du nœud est égal au nombre de* 1*).*

`Initialisation :`
La propriété P_0 est vraie.
Supposons P_n vraie et montrons P_{n+1}. Considérons l'arbre B_{n+1} de même que précédemment l'arbre B_{n+1} a un fils identique à B_n au préfixe 0 de ses clefs près et l'arbre B_{n+1} privé de son fils gauche est identique à B_n au préfixe 1 de ses clefs près.
Par hypothèse de récurrence le degré des nœuds du fils gauche de B_{n+1} vérifie P_n, le préfixe 0 ne changeant pas le nombre de 1 à droite du 0 le plus à droite.
Les nœuds de B_{n+1} privés de son fils gauche, par hypothèse de récurrence vérifient P_n, au préfixe 1 près.
D'où les nœuds de B_{n+1} vérifient P_{n+1}
La propriété P_n étant vraie au rang 0 et étant héréditaire elle est vraie à tout rang $n \in mathbbN$.

1.2 Coût amorti

Version de Philippe
La fonction de potentiel Φ, relativement à l'état initial *Arbre vide* est une fonction de potentiel valide, puisque toujours plus grande que sa valeur à l'etat initial.
Calculons maintenant le coût amorti par la méthode du potentiel de la fonction d'insertion avec éclatements à la descente. On a

$$\hat{C} = c_{reel} + \Phi_{apres} - \Phi_{avant}$$

Le coût réel de l'insertion est le suivant : on paye $O(1)$ à chaque niveau pour descendre plus éventuellement un surcoût lié à l'éclatement du nœud que l'on traverse. Ce surcoût est constant. Soit un coût réel :

$$c_{reel} = O(log(n)) + nb\_eclatement$$

Calculons la diffférence de potentiel entre létat intial et létat final. À chaque éclatement, on supprime un 4-nœud et on crée un 3-nœud (au niveau supérieur). On a donc une variation de potentiel égale à l'opposé du nombre d'éclatements.
Soit un coût amorti de $O(log(n))$ pour l'insertion avec éclatements à la descente.

Version de Guénaël
Dans les structures de données, il arrive parfois que, pour optimiser la complexité sur un grand nombre d'opérations, il faille régulièrement faire des *opérations de maintenance*. Calculer avec le coût amorti revient à répartir l'impact (pour la complexité) de ces opérations de maintenance, généralement coûteuses, sur les opérations qui bénéficient de cette maintenance.
Il faut d'abord bien comprendre ce qu'est le coût amorti exprimé à partir d'une focntion de potentiel :

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

pour la i-ième opération, le coût amorti est a_i, le coût réel est c_i et la *différence de potentiel entre cette opération (la i-ième) et la précédente (la $(i-1)$-ième)* est $\Phi_i - \Phi_{i-1}$.
Important : Le coût amorti sera une borne du coût réel lorsque $\Phi_i - \Phi_0 \geqslant 0$. La fonction de potentiel proposée ici vérifie $\Phi_0 = 0$ et $\Phi_i \geqslant 0$ pour tout $i > 0$, le coût amorti calculé ci-après sera donc une borne du coût réel.
Dans cet exercice, l'opération couteuse de maintenance, c'est l'éclatement des 4-nœuds ; cette opération est coûteuse, mais elle n'a besoin d'être faite que de temps en temps, et surtout, elle permet de diminuer le coût de l'insertion.
Attention, la complexité se calcule ici en nombre d'éclatements, et non pas, comme il est généralement le cas, en nombre de comparaisons.

Pour traiter correctement cet exercice, il faut bien avoir en tête l'algorithme d'éclatement à la descente (et ne pas le confondre avec son dual, qui est l'éclatement en remontée). L'algorithme est décrit dans les transparents du cours de M. Soria ; si vous n'avez toujours pas lu ce document, il est temps de le faire !
Le principe d'éclatement à la descente est le suivant. Lors de l'ajout d'un élément, l'ajout se fait aux feuilles ; comme l'ajout se fait aux feuilles, il faut descendre dans l'arbre, et à chaque fois que nous rencontrons un 4-nœud, nous l'éclatons. Finalement, après avoir fait tous les éclatements nécessaires, nous nous retrouvons dans une feuille, et nous y ajoutons l'élément.
Nous insistons sur le fait que, dans l'algorithme d'éclatement à la descente, lorsque nous considérons l'éclatement du nœud x, le père $p(x)$ du nœud x ne peut pas être un 4-nœud. En effet, comme nous éclatons à la descente, avant de descendre dans le nœud x, nous avons été dans le nœud $p(x)$ et si celui-ci était un 4-nœud, nous l'avons, depuis, éclaté. Ceci est très important.
Nous pouvons donc considérer que l'insertion d'un élément e se fait en deux temps :

1. d'abord, nous partons de la racine, et en descendant jusqu'à feuilles, nous faisons éclater tous les nœuds dont il y a besoin ;
2. une fois que nous nous trouvons dans la bonne feuille, nous ajoutons l'élément e à cette feuille.

Pour le premier temps, la descente, lorsque nous avons besoin d'éclater un nœud y, il faut considérer deux cas :
– soit le père $p(y)$ de y est un 2-nœud, auquel cas, d'après la transformation qui est décrite dans le transparent 19 :

$$P_2 =< p(y) = [x], A_1, A_2 > \quad \text{avec} \quad A_1 =< y = [a, b, c], U_1, U_2, U_3 >$$

devient :

$$P_2' =< [b, x], < [a], U_1, U_2 >, < [c], U_3, U_4 >, A_2 > ,$$

nous voyons que pour passer de P_2 à P_2', nous avons supprimé 4-nœud et formé 3-nœud (et formé des 2-nœuds, mais ça, en ce qui concerne la fonction de potentielle, ce n'est pas important) ; ainsi l'éclatement a entraîné une diminution du potentiel de 1 (en effet, la suppression du 4-nœud fait baisser le potentiel de 2, et la création du 3 nœud le fait augmenter de 1) ;
– soit le père $p(y)$ de y est un 3-nœud, auquel cas, d'après la transformation qui est décrite dans le transparent 19 :

$$P_3 =< p(y) = [a, b], A_1, A_2, A_3 > \quad \text{avec} \quad A_2 =< y = [c, d, e], U_1, U_2, U_3, U_4 >$$

devient :

$$P_3' =< [a, d, b], A_1, < [c], U_1, U_2 >, < [e], U_3, U_4 >, A_3 >$$

et nous voyons que pour passer de P_3 à P_3', nous avons supprimé le 4-nœud (y) et le 3-nœud père ($p(y)$), et à la place nous avons formé un 4-nœud. Ainsi, l'éclatement a entraîné une diminution du potentiel de 1. (Encore une fois : suppression d'un 4-nœud, donc baisse de 2, suppression d'un 3-nœud, donc baisse de 1, et création d'un 4-nœud donc augmentation de 2 ; ce qui fait au final, $-2 - 1 + 2 = -1$, donc une diminution du potentiel de 1).
– comme nous l'avons déjà dit, le père $p(y)$ ne peut pas être un 4-nœud (nous l'aurions déjà éclaté).
Ce que nous constatons donc :***chaque*** **éclatement entraîne une diminution du potentiel de** 1.

Pour le second temps (plus simple), considérons l'ajout d'un élément e à une feuille, plusieurs cas sont possibles :
– soit cette feuille est un 2-nœud, auquel cas, rajouter un élément forme un 3-nœud (et donc entraîne une augmentation du potentiel de 1) ;
– soit cette feuille est un 3-nœud, auquel cas, rajouter un élément à cette feuille forme un 4-nœud, tout en faisant disparaître le 3-nœud qui était là à l'origine (cela entraine donc une augmentation du potentiel de 1 ; car suppression du 3-nœud, et création du 4-nœud) ;
– la feuille ne peut pas être un 4-nœud car, là encore, nous l'aurions éclatée (l'éclatement à la descente va jusqu'au feuilles).
Au final, nous constatons que **l'insertion d'un élément dans une feuille entraîne une augmentation du potentiel de** 1.

Maintenant, nous allons combiner ces deux étapes. Pour cela, il nous faut calculer la formule :

$$a_i = c_i + \Phi_i - \Phi_{i-1}.$$

Le coût amorti de la i-ième insertion est c_i, le nombre d'éclatements réalisés au cours de cette insertion, auquel nous ajoutons la différence de potentiel entraînée par cette insertion.
Nous avons vu que la différence de potentiel $\Phi_i - \Phi_{i-1}$ est donnée par :
– -1 (une diminution de 1) pour **chaque** éclatement réalisé pour l'insertion i ;
– 1 pour l'ajout final de l'élément aux feuilles.
Ainsi, $\Phi_i - \Phi_{i-1} = 1 + -1 \cdot c_i$ (rappelons une dernière fois que c_i est, par définition, puisque nous mesurons la complexité en nombre d'éclatements, le nombre d'éclatements entrainés à la i-ième insertion). Quand

nous replaçons ça dans l'équation de coût amorti on obtient :

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= c_i + (1 + -1 \cdot c_i) \\ &= c_i + (1 - c_i) \\ &= c_i + 1 - c_i \\ &= 1 \end{aligned}$$

Ce qui nous permet de déduire que le coût amorti de la i-ième insertion est 1 : nos insertions se font donc en $\mathcal{O}(1)$.

Une question intéressante à poser aux étudiants : pourquoi est-ce possible d'avoir des insertions en $\mathcal{O}(1)$? ceci ne viole-t-il pas la borne inférieure pour le tri ?

Version de Maryse
Potentiel $\Phi(A) = 2n_4 + n_3$.
Il y a ε (valant 0 ou 1) éclatements à la racine.
Il y a k_1 éclatements de 4-nœuds fils de 2-nœuds.
Il y a k_2 éclatements de 4-nœuds fils de 3-nœuds.
Il y a en tout $k_1 + k_2 + \varepsilon$ éclatements. Le coût réel est donc $c = k_1 + k_2 + \varepsilon$.
Calcul du nouveau potentiel
Pour chaque éclatement d'un 4-nœud fils d'un 2-nœud, il y a un 3-nœud en plus et un 4-nœud en moins ; après les k_1 éclatements de 4-nœuds fils de 2-nœuds, il y a donc k_1 3-nœuds en plus et k_1 4-nœuds en moins.

Pour chaque éclatement d'un 4-nœud fils d'un 3-nœud, il y a un 3-nœud en moins et le même nombre de 4-nœuds ; après les k_2 éclatements de 4-nœuds fils de 2-nœuds, il y a donc k_2 3-nœuds en moins et le même nombre de 4-nœuds.

Si la racine est un 4-nœud, il y a 4-nœud en moins ; après les ε éclatements de la racine, il y a donc ε 4-nœuds en moins.

L'insertion se fait dans une feuille qui est soit un 2-nœud soit un 3-nœud. Il y a ε_1 insertions dans une feuille qui est un 2-nœud et ε_2 insertions dans une feuille qui est un 3-nœud, avec ε_1 valant 0 ou 1 et ε_2 valant $1 - \varepsilon_1$. Si l'insertion se fait dans un 2-nœud, il y a un 3-nœud en plus ; si l'insertion se fait dans un 3-nœud, il y a un 3-nœud en moins et un 4-nœud en plus. Pour l'insertion dans une feuille, le nombre de 3-nœuds varie donc de $\varepsilon_1 - \varepsilon_2$ et le nombre de 4-nœuds varie de ε_2.

Au total, le nombre de 3-nœuds varie de $k_1 - k_2 + \varepsilon_1 - \varepsilon_2$ et le nombre de 4-nœuds varie de $-k_1 - \varepsilon + \varepsilon_2$.
Le nouveau potentiel est donc : $\Phi(A') = 2(n_4 - k_1 - \varepsilon + \varepsilon_2) + (n_3 + k_1 - k_2 + \varepsilon_1 - \varepsilon_2)$
$\Phi(A') = 2n_4 + n_3 - 2\varepsilon - k_1 - k_2 + \varepsilon_1 + \varepsilon_2 = 2n_4 + n_3 - 2\varepsilon - k_1 - k_2 + 1$ (puisque $\varepsilon_1 + \varepsilon_2 = 1$).
Calcul du coût amorti
La différence de potentiel est : $\Phi(A') - \Phi(A) = -k_1 - k_2 - 2\varepsilon + 1$
et le coût amorti est : $\hat{c} = c + \Phi(A') - \Phi(A) = k_1 + k_2 + \varepsilon - k_1 - k_2 - 2\varepsilon + 1 = 1 - \varepsilon$.
Donc $\hat{c} \leq 1$ et le coût amorti est en $O(1)$.

2 Partiel de novembre 2005 (extrait)

2.1 Tas et arbre binômial

On utilisera les primitives suivantes sur les arbres binaires :
Clef(t) qui renvoie la clef à la racine de t
Estfeuille(t) qui renvoie si t est réduit à une feuille ou non
FilsG(t) qui renvoie le fils gauche de t.
FilsGG(t) qui renvoie le fils gauche du fils gauche de t.
FilsGD(t) qui renvoie le fils droit du fils gauche de t.

FilsD(t) qui renvoie le fils droit de t.
Arbre(x, G, D) qui renvoie un arbre dont la clef à la racine est x et qui a G (resp. D) comme sous-arbre gauche (resp. droit).
Vide renvoie un arbre vide.

Et les primitives suivantes pour les arbres binomiaux :
Feuille(x) qui crée un arbre binomial réduit à une feuille avec comme clef x.
Ajout(X, Y) qui prend deux arbres binomiaux et qui rajoute X comme fils de la racine de Y.

Question 2.1.1 :

Algorithme 1 : Glissement

Entrées : Un arbre binaire t
Sorties : Un tas binaire

```
1 Glissement(t)
2 t' :=Arbre(Clef(t),FilsD(t),Vide)
3 retourner GlisserAux(t')
```

Algorithme 2 : GlisserAux

Entrées : Un arbre binaire t
Sorties : Un tas binaire

```
1 GlisserAux(t)
2 si Estfeuille(t) alors
3 |  retourner t
4 sinon
5 |  retourner Arbre(Clef(t),GlisserAux(FilsG(t),FilsGD(t))
```

Le nombre $N(t)$ d'appel à la primitive Arbre dans GlisserAux vérifie : $N(t) = 1 + N(FilsG(t))$. On a donc $O(k)$ appels.
Question 2.1.2 :

Algorithme 3 : Bino

Entrées : Un tas binaire
Sorties : Un tas binomial

```
1  Bino(t)
2  si Estfeuille(t) alors
3  |  retourner Feuille(Clef(t))
4  sinon
5  |  X :=FilsG(t)
6  |  Y :=Arbre(Clef(t),Vide,FilsD(t))
7  |  Y :=Glissement(Y) (*remarque : c'est stupide de mettre le fils à droit à la ligne du dessus
   |  pour ensuite le mettre à gauche dans Glissement*)
8  |  A :=Bino(X)
9  |  B :=Bino(Y)
10 |  retourner Ajout(A,B) (*remarque fondamentale : à cause du glissement on a toujours
   |  clef(X)>clef(Y)*)
```

On obtient avec l'arbre donné : 1(3(8(11(15),14)),7(13),16)),2(5(12),6),4(10),9)
niveau 1 : 1
niveau 2 : 3 2 4 9
niveau 3 : 8 7 16 5 6 10
niveau 4 : 11 14 13 12

niveau 5 : 15

Question 2.1.3 :

Le nombre $N(t)$ d'appels à la primitive Ajout dans Bino vérifie : $N(t) = 1 + N(FilsG(t)) + N(FilsD(t))$. On a donc $O(2^k)$ appels.

2.2 Coût amorti

Question 1
On montre qu'à tout moment :
(a) les éléments de P_1 sont empilés du plus ancien dans la structure F au plus récent
(b) les éléments de P_2 sont empilés du plus récent dans la structure F au plus ancien
(c) tout élément de P_2 est plus ancien dans la structure F que tout élément de P_1.
Supposons que ce soit vrai à un certain moment.
Si on ajoute un élément x à F alors on empile x dans P_1 et il est clair que (a), (b) et (c) sont encore vérifiés.
Si on enlève un élément x à F alors :
Cas 1 : P_2 n'est pas vide
On dépile P_2, il est clair que (a), (b) et (c) sont encore vérifiés ;
remarque : on dépile l'élément le plus ancien dans la structure F.
Cas 2 : P_2 est vide
Dans P_1 on a $x_1, x_2, \ldots, x_k$ empilés dans cet ordre (du plus ancien au plus récent). On dépile tous les éléments de P_1 et on les empile dans P_2 au fur et à mesure. Dans P_2 on obtient $x_k, \ldots, x_2, x_1$ empilés dans cet ordre (du plus récent au plus ancien). On dépile P_2, il est clair que (a), (b) et (c) sont encore vérifiés ;
remarque : on dépile l'élément le plus ancien dans la structure F.

Question 2
On calcule le coût réel :
- d'un ajout $c(ajout) = 1$
- d'un retrait $c(retrait) = 2\,|P_1| + 1$.

Dans le calcul du coût amorti, on note F la file avant l'opération et F' la file après l'opération.
a) Essai 1 : $\Phi_1(F) = 2\,|P_1|$
Coût amorti d'un ajout : $\Phi_1(F') = 2(|P_1| + 1)$, $\hat{c_1}(ajout) = 3$.
Coût amorti d'un retrait :
Cas 1 P_2 n'est pas vide : $\Phi_1(F') = 2\,|P_1|$, $\hat{c_1}(retrait) = 1$.
Cas 2 P_2 est vide : $\Phi_1(F') = 0$, $\hat{c_1}(retrait) = 2\,|P_1| + 1 - 2\,|P_1| = 1$.
Le potentiel Φ_1 est toujours positif ou nul, le coût réel total est donc toujours inférieur ou égal au coût amorti total.
b) Essai 2 : $\Phi_2(F) = |P_1| - |P_2|$
Coût amorti d'un ajout : $\Phi_1(F') = |P_1| + 1 - |P_2|$, $\hat{c_2}(ajout) = 2$.
Coût amorti d'un retrait :
Cas 1 : P_2 n'est pas vide : $\Phi_1(F') = |P_1| - (|P_2| - 1)$, $\hat{c_2}(retrait) = 2$.
Cas 2 : P_2 est vide : $\Phi_2(F) = |P_1|, \Phi_2(F') = -(|P_1| - 1)$, $\hat{c_2}(retrait) = 2$.
Le potentiel Φ_2 est parfois négatif, donc le coût amorti total ne majore pas toujours le coût réel total. Par exemple, si on empile n éléments et si on en dépile 1 ensuite, le coût réel total est $3n + 1$ et le coût amorti total est $2n + 2$.

2.3 Arbres bicolores

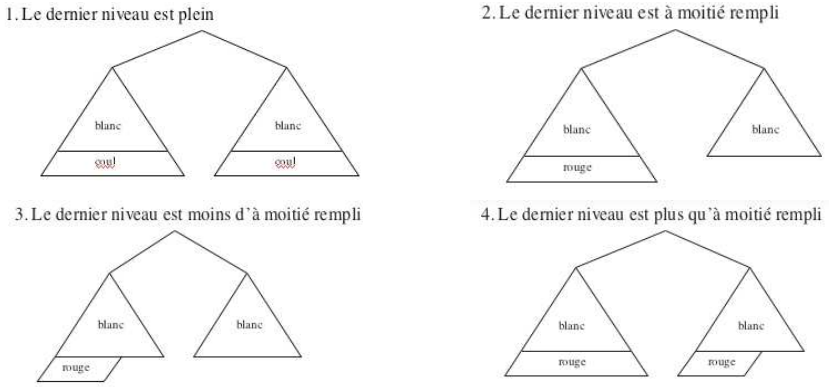
Dans un arbre bicolore, j'appelle *feuille étiquetée* tout nœud dont les deux fils sont des feuilles (blanches, non étiquetées). Dans la définition de la hauteur d'un arbre bicolore, je pose $h(B) = 0$ si B est une feuille (blanche, non étiquetée).

Question 1

Soit $L = (x_1, \ldots, x_n)$ une liste triée de longueur n. L'idée (une idée possible) est de construire un arbre parfait dont tous les nœuds sont blancs, sauf éventuellement les feuilles étiquetées de plus bas niveau. L'arbre parfait a une hauteur h telle que $2^{h-1} \leq n < 2^h$, *i.e.* $h = \lfloor \log_2 n \rfloor + 1$ et son dernier niveau a un nombre de feuilles inférieur ou égal à 2^{h-1}.

Appelons $BicP(L)$ l'arbre bicolore parfait correspondant à la liste L.

Quatre cas peuvent se produire, illustrés par les dessins suivants :



Premier cas ($n = 2^h - 1$) : on construit un arbre complet dont tous les nœuds sont blancs, exception faite des feuilles étiquetées qui sont ou bien tous blanches ou bien toutes rouges. Appelons $BicC(L, coul)$ l'arbre complet dont les feuilles étiquetées sont toutes de la couleur *coul* et qui correspond à la liste L. Alors $BicC(L, coul)$ est l'arbre bicolore complet dont la racine est blanche d'étiquette x_k, dont le fils gauche est $BicC(L_1, coul)$ et dont le fils droit est $BicC(L_2, coul)$, avec $k = \frac{n+1}{2}$, $L_1 = (x_1, \ldots, x_{k_1})$ et $L_2 = (x_{k+1}, \ldots, x_n)$.

On peut décider que $BicP(L) = BicC(L, blanc)$.

Deuxième cas ($n = 3.2^{h-2} - 1$) : $BicP(L)$ est l'arbre bicolore parfait dont la racine est blanche d'étiquette x_k, dont le fils gauche est $BicC(L_1, rouge)$ et dont le fils droit est $BicC(L_2, blanc)$, avec $k = 2^{h-1} = \frac{2(n+1)}{3}$, $L_1 = (x_1, \ldots, x_{k_1})$ et $L_2 = (x_{k+1}, \ldots, x_n)$.

Troisième cas ($n < 3.2^{h-2} - 1$) : $BicP(L)$ est l'arbre bicolore parfait dont la racine est blanche d'étiquette x_k, dont le fils gauche est $BicP(L_1)$ et dont le fils droit est $BicC(L_2, blanc)$, avec $k = n - (2^{h-1} - 1)$, $L_1 = (x_1, \ldots, x_{k_1})$ et $L_2 = (x_{k+1}, \ldots, x_n)$.

Quatrième cas ($n > 3.2^{h-2} - 1$) : $BicP(L)$ est l'arbre bicolore parfait dont la racine est blanche d'étiquette x_k, dont le fils gauche est $BicC(L_1, rouge)$ et dont le fils droit est $BicP(L_2)$, avec $k = 2^{h-1}$, $L_1 = (x_1, \ldots, x_{k_1})$ et $L_2 = (x_{k+1}, \ldots, x_n)$.

Estimation du coût

En notant $c(h)$ le coût pour construire le bicolore parfait de hauteur $h = \lfloor \log_2 n \rfloor + 1$ correspondant à une liste L de longueur n, on obtient $c(h) \leq \alpha + 2c(h)$ où α est une constante. Le coût est donc en $O(2^h)$ et donc en $O(n)$.

Question 2

```
Algo SupprimerMarques(B : ArbreBicolore)
L <- LNM(B)
B <- BicP(L)
```

$LNM(B)$ renvoie la liste infixe des nœuds non marqués de B :

```
Fonction LNM(B : ArbreBicolore) : Liste
si EstFeuille(B) alors
  Retourne ListeVide
sinon
  e <- CleRacine(B) ; G <- SousArbreGauche(B) ; D <- SousArbreDroit(B)
  si e a une marque alors Retourne Concatenation (LNM(G),LNM(D))
  sinon Retourne Concatenation (LNM(G),(e),LNM(D)) fin si
fin si
```

$BicP(L)$ renvoie l'arbre bicolore parfait correspondant à la liste L :

```
Fonction BicP(L : Liste) : ArbreBicolore
si EstListeVide(L) alors
  Retourne FeuilleBlanche
sinon
  n <- Longueur(L) ; h <- partie entiere de log2(n)
  si n = 2^h - 1 alors
    Retourne BicC(L,blanc)
  sinonSi n < 3*2^(h-2) - 1 alors
    k <- n + 1 - 2^(h-1) ; r <- L[k] ; L1 < L[1..k-1] ; L2 <- L[k+1..n]
    Retourne ArbreBinaire(r,blanc,BicP(L1),BicC(L2,blanc))
  sinon
    k <- 2^(h-1)  ; r <- L[k] ; L1 < L[1..k-1] ; L2 <- L[k+1..n]
    Retourne ArbreBinaire(r,blanc,BicC(L1,rouge),BicP(L2))
  fin si
fin si
```

$BicC(L, coul)$ renvoie l'arbre bicolore complet dont les feuilles étiquetées sont toutes de la couleur *coul* et qui correspond à la liste L, sous l'hypothèse que n est de la forme $2^h - 1$:

```
Fonction BicC(L : Liste, coul : couleur) : ArbreBicolore
si EstListeVide(L) alors
  Retourne FeuilleBlanche
sinon
  n <- Longueur(L)
  si n = 1 alors
    Retourne  ArbreBinaire(r,coul,FeuilleBlanche,FeuilleBlanche)
  sinon
    k <- (n+1)/2 ; r <- L[k] ; L1 < L[1..k-1] ; L2 <- L[k+1..n]
    Retourne ArbreBinaire(r,blanc,BicC(L1,coul),BicC(L2,coul))
  fin si
fin si
```

Chacune des deux fonctions LNM et $BicP$ est en $O(n)$ donc l'algorithme de suppression des nœuds marqués est en $O(n)$.

Question 3
Puisque chaque opération s'effectue sur un arbre bicolore de taille au plus $\frac{3n}{2}$, elle coûte au plus $\beta \log n$ où β est une constante. Tant que l'on n'a pas fait la totalité des $n/2$ opérations, il ne peut pas y avoir au moins la moitié des nœuds marqués. Il y a donc au plus une reconstruction de l'arbre, à la fin. Le coût total est donc majoré par $(n/2) * \beta \log n + \gamma n$. Il est donc en $O(n \log n)$.