UPMC SORBONNE UNIVERSITÉS

Structures de données avancées
TD Semaines 1 – 3

Version du 20 septembre 2012

1 Rappels : notations asymptotiques

Exercice 1 – Notations de Landau

1. Rappeler les définitions des notations de Landau, O, Ω et Θ, pour les fonctions à valeur dans $\mathbb{R}_+^*$.
2. 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 : $\forall x \in \mathbb{R},\ max(f,g)(x) = max(f(x), g(x))$.
3. Montrer que $O(f+g) = O(max(f,g))$.
4. Montrer que :
$$\sum_{p=1}^{n} p^k = \Theta(n^{k+1}).$$

Exercice 2 – Complexité dans le pire des cas

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 ?
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 ?
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 ?
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 ?

Exercice 3 – Classement de fonctions

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

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

©2011-2012 (by UPMC/Master STL/AlgoAv)

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

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

Exercice 4 – 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.

1. Calculer le coût amorti d'une opération en utilisant la méthode par agrégat.
2. Même question en utilisant la méthode du potentiel.

Exercice 5 – 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).

1. En utilisant la méthode par agrégat.
2. En utilisant la méthode du potentiel.

©2011-2012 (by UPMC/Master STL/AlgoAv)

Exercice 6 – Coût amorti (tableaux dynamiques)

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.

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

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.

Exercice 7 – 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 `Incrémenter(A)` comme suit :

```
Incrémenter(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
```

1. Quel est l'effet de `Incrémenter` sur le tableau A ?
2. Le coût réel de l'opération `Incrémenter` 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 `Incrémenter` :
   (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).

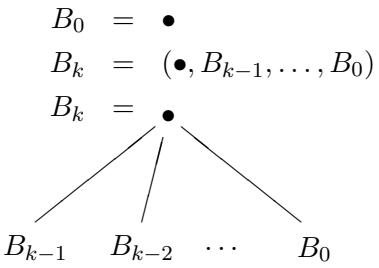
3 Files de Fibonacci

Exercice 8 – Définitions et propriétés des arbres binomiaux

On rappelle deux définitions équivalentes des *arbres binomiaux*, qui ont été introduits dans le premier cours.

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.



1. Prouver d'abord que ces deux définitions sont équivalentes (un arbre obtenu à partir de la première définition satisfait la seconde définition, et *vice-versa*).
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 [1] 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.

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.

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

Exercice 9 – 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.

1. (a) 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 ?
   (b) 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 : $(B_3, B_1, B_2, B_1, B_1, B_2, B_4, B_0, B_0, B_0)$.

2. Montrer que le degré maximum d'un nœud d'une file binomiale relâchée est aussi $D(n)$.

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.

3. Écrire les procédures
   (a) `ConcaténerFBR` d'union de deux files binomiales relâchées (en une file binomiale relâchée). Quelle est la complexité ?
   (b) `InsérerFBR` d'insertion d'une nouvelle valeur v dans une file binomiale relâchée. Quelle est la complexité ?
4. (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).
5. 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). Remarquer que lorsqu'on supprime des nœuds dans une file H, il se peut que $\min(H)$ ne soit plus positionné sur la racine de l'arbre de plus petite clé. C'est pourquoi on considère qu'on dispose d'une procédure `MettreAJourMin` qui remet à jour $\min(H)$.
6. En utilisant la méthode du potentiel, calculer 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.

Exercice 10 – Nombres de Fibonacci (prépare le calcul de coût amorti de l'exercice suivant)

On rappelle que les nombres de Fibonacci sont définis par : $F_0 = 0$, $F_1 = 1$ et $F_{k+2} = F_{k+1} + F_k$ pour $k \geqslant 0$.

1. Montrer que les nombres de Fibonacci vérifient l'égalité $F_{k+2} = 2 + \sum_{i=2}^{k} F_i$ lorsque $k \geqslant 2$.
2. Montrer par récurrence que $F_{k+2} \geqslant \Phi^k$ avec $\Phi = \dfrac{1+\sqrt{5}}{2}$ (sachant que $\Phi \sim 1,61803$ est tel que $\Phi^2 - \Phi - 1 = 0$).

Exercice 11 – Files de Fibonacci

Dans l'exercice 6, en relâchant une contrainte sur la composition de la file binomiale, il a été possible d'améliorer la complexité des opérations d'union et d'insertion de $O(\log n)$ à $O(1)$.

On cherche maintenant à améliorer la complexité de l'opération de diminution de clef qui, avec les files binomiales (relâchées ou pas), se fait en $O(\log n)$.

Pour cela, on part de la structure de file binomiale relâchée. Lorsque la diminution de la clef du nœud x viole la propriété de croissance (c'est-à-dire, lorsque la clef diminuée du nœud x est plus grande que la clef de son parent $p(x)$), au lieu de corriger l'arbre, on coupe le sous-arbre enraciné en x, et on le rajoute à la file.

Dans cet exercice, on peut réutiliser les primitives algorithmiques définies précédemment.

1. Quelle est la complexité de l'opération de diminution de clef ainsi réalisée ? Quelle est sa conséquence pour la structure de la file ?

On appelle *file de Fibonacci*, une file binomiale relâchée dans laquelle on autorise tout nœud x d'une file, *qui n'est pas une racine*, à avoir un fils de moins qu'il ne devrait depuis la dernière fois qu'il a été attaché à son père, avec un marquage de cette situation ; on autorise de plus une racine à perdre un nombre arbitraire de fils.

2. Chaque nœud x contient un booléen `marque` (x). Lorsque x n'est pas une racine, `marque` (x) a la valeur vrai s'il manque un fils à x et faux sinon. Les marques des racines sont positionnées à faux.
   (a) Écrire la procédure `DiminuerClefFib` de diminution d'une clef dans une file de Fibonacci.
   (b) Pourquoi est-ce que l'on n'exige pas que les racines aussi ne perdent qu'au plus un fils ? ($\star$)
3. En utilisant pour fonction de potentiel

$$\Phi(H) := a(H) + 2m(H)$$

où $a(H)$ est le nombre d'arbres dans la file H et $m(H)$ le nombre de marques à vrai, déterminer le coût amorti de l'opération `DiminuerClefFib`.

Démontrer les deux lemmes ci-dessous aboutissant à une minoration des degrés des nœuds des arbres composants une file de Fibonacci.

4. **Lemme 1**. Soit N un nœud d'une file de Fibonacci tel que `degre` $(N) = k$. Soient $x_1, \ldots, x_k$ ses fils dans l'ordre où ils ont été liés à N. Alors `degre` $[x_1] \geqslant 1$ et `degre` $[x_i] \geqslant i - 2$ si $2 \leqslant i \leqslant k$.
5. **Lemme 2**. Soit N un nœud d'une file de Fibonacci tel que `degre` $(N) = k$, le nombre $taille(N)$ de nœuds du sous-arbre de racine N (N compris) est minoré par F_{k+2}.
6. On note $d(n)$ le degré maximum d'un nœud dans une file de Fibonacci de taille n. Déduire de ce qui précède que $d(n) \leqslant \log_\Phi n$.
7. Proposer deux algorithmes pour supprimer dans une file de Fibonacci H un nœud x donné de clef k.
   Par la méthode du potentiel, calculer le coût amorti de la suppression d'une clef dans une file de Fibonacci.

©2011-2012 (by UPMC/Master STL/AlgoAv)

4 Arbres bicolores et arbres 2-3-4

Exercice 12 – Arbres 2-3-4

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

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 :

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

Primitives

Pour manipuler les arbres bicolores, on dispose 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 de primitives spécifiques aux arbres bicolores :

`FeuilleBlanche`	*Rien* $\rightarrow$ *arbre binaire*
`FeuilleBlanche()`	*renvoie l'arbre binaire réduit à une feuille de couleur c blanche*
`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*

Exercice 13 – 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.

1. Illustrer le principe énoncé ci-dessus (au moyen de petits dessins).
2. Montrer que l'arbre ainsi obtenu est un arbre bicolore et encadrer la hauteur de cet arbre.
3. Construire un arbre bicolore, que l'on nommera `B-ex1`, transformé de l'arbre 2-3-4 `A-ex1`.
4. 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.

Exercice 14 – Transformation d'un arbre bicolore en arbre 2-3-4

1. Donner le principe de la transformation d'un arbre bicolore en arbre 2-3-4. Illustrer ce principe au moyen d'un exemple.
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 d'éléments et une liste d'arbres 2-3-4.

Exercice 15 – Rotations dans un arbre binaire

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

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

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

`RotationDroiteGauche`	*arbre binaire → arbre binaire*
`RotationDroiteGauche(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*
`RotationGaucheDroite :`	*arbre binaire → arbre binaire*
`RotationGaucheDroite(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*

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

Exercice 16 – 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.

1. Transposer dans les arbres bicolores les différents cas d'insertion d'une clef dans une feuille d'un arbre 2-3-4.
2. Transposer dans les arbres bicolores les différents cas d'éclatement d'un 4-nœud.
3. Réaliser l'insertion des clefs 13.1, 13.3, 12.5, 12.8 dans l'arbre bicolore `B-ex1`.
4. Écrire l'algorithme d'insertion d'une clef dans un arbre bicolore.
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
```

©2011-2012 (by UPMC/Master STL/AlgoAv)