université de BORDEAUX

Année Universitaire 2015/2016
Session 1 d'automne

Parcours / Étape : IM300 & IN301
Code UE : J1IN3W01
Épreuve : Algorithmique et structures de données 1
Date : Mardi 5 janvier 2016 **Heure :** 11 heures
Durée : 1 heure 30
Documents : non autorisés
Épreuve de M. Alain Griffault

Collège Sciences et Technologies

SUJET + CORRIGE

Avertissement

- La plupart des questions sont indépendantes.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).
- Le sujet est sans doute un peu long.
- La note maximale est de 23/20 → 20/20.

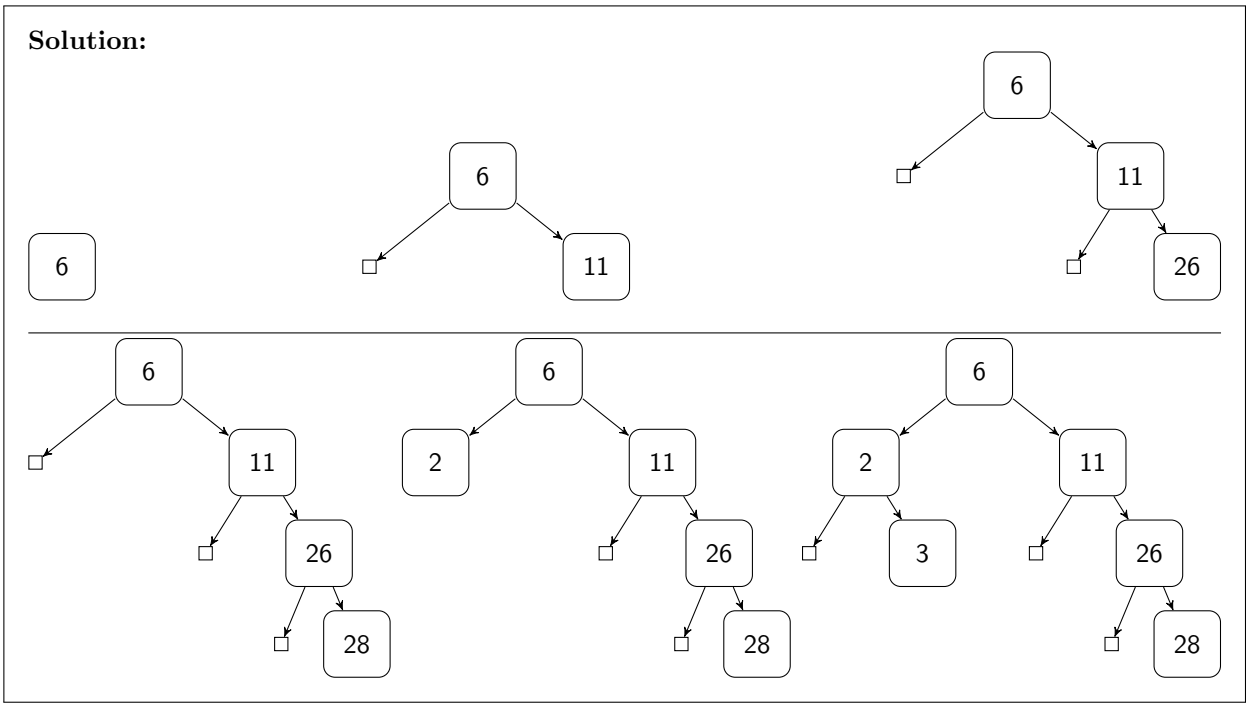
Question	Points	Score
Insertions dans les ABR, Tas et AVL	4	
L'évaluation d'expressions	7	
Les arbres rouge et noir	12	
Total:	23	

Exercice 1 : Insertions dans les ABR, Tas et AVL (4 points)

Soit la liste de clés `L = (6, 11, 26, 28, 2, 3)`. Pour chacune des structures, en partant d'un arbre binaire vide, vous devez dessiner l'arbre après chacune des insertions des éléments de la liste L.

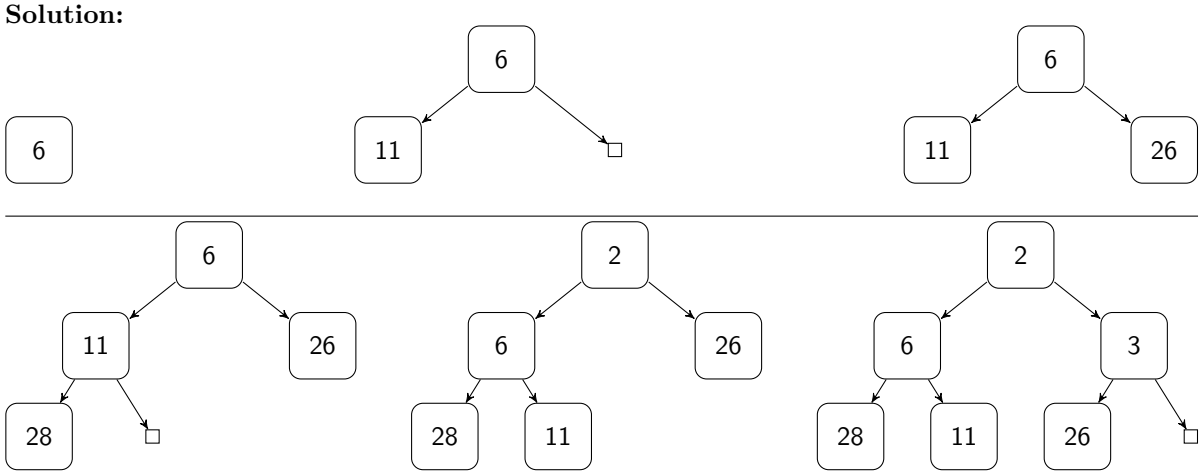
(a) (1 point) Arbre binaire de recherche (ABR) : insertion de `L = (6, 11, 26, 28, 2, 3)`.

Solution:



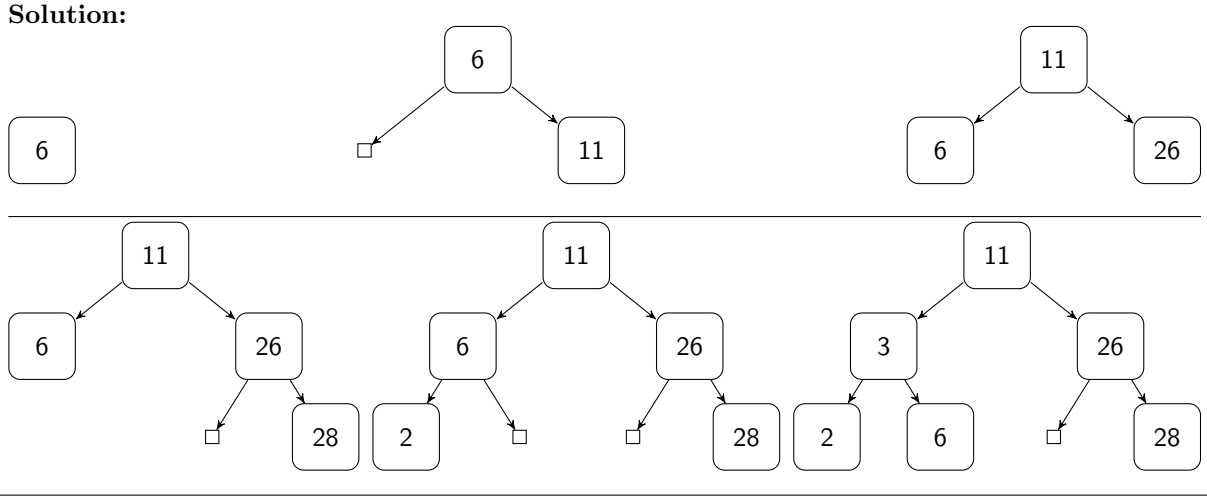
(b) (1½ points) Tas min : insertion de `L = (6, 11, 26, 28, 2, 3)`.

Solution:



(c) (1½ points) AVL : insertion de `L = (6, 11, 26, 28, 2, 3)`.

Solution:



Exercice 2 : L'évaluation d'expressions (7 points)

L'objectif est l'écriture d'un algorithme qui évalue une expression arithmétique bien parenthésée à l'aide d'une pile. Informellement, les expressions arithmétiques considérées n'utilisent que l'addition, notée +, la soustraction, notée − et la multiplication, notée ×. Par exemple :

Expression	Bien parenthésée	Évaluation	Explication
356	Vrai	356	
$(5+15)$	Vrai	20	
$(((5+15)\times(-3+6))-5)$	Vrai	55	
(356)	Faux	-	Trop parenthésée
$5+15$	Faux	-	Pas suffisamment parenthésée
$(5+15+10)$	Faux	-	Pas suffisamment parenthésée
$(5+15\times 10)$	Faux	-	Pas suffisamment parenthésée
$(5+c)$	Faux	-	c n'est pas un nombre
$(5+15($	Faux	-	Mauvais parenthésage

Les expressions sont stockées dans des tableaux d'objets. Ainsi $(((5+15)\times(-3+6))-5)$ est représentée par :

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
valeur	(	(	(	5	+	15	)	×	(	-3	+	6	)	)	-	5	)

Définition : Une expression arithmétique E est **bien parenthésée** si et seulement si :

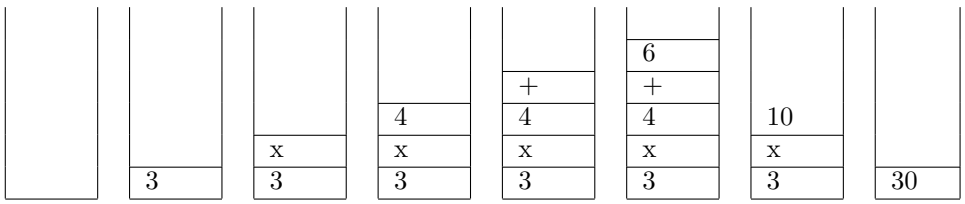
$$E = \begin{cases} \text{Un nombre} \\ \text{ou bien} \\ (E1 \text{ op } E2) \end{cases}$$

avec $E1$ et $E2$ deux expressions bien parenthésées et "op" un opérateur arithmétique binaire.

Pour évaluer une expression bien parenthésée à l'aide d'une pile, les principes sont les suivants :
- Le tableau contenant l'expression est parcouru itérativement.
- En fonction du type de chaque élément, le traitement diffère.
  - Les opérateurs et les nombres sont stockés dans la pile.
  - Les parenthèses ouvrantes sont ignorées.
  - Les parenthèses fermantes *déclenchent* l'unique calcul possible, et stocke le résultat de ce calcul dans la pile.
- À la fin de l'itérative :
  - Si l'expression est une expression bien parenthésée, le résultat est l'unique objet contenu dans la pile.
  - La réciproque n'est pas vraie.

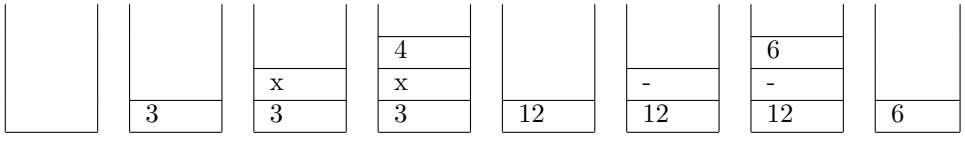
Avec l'aide de compteurs, il est possible de déterminer lors de l'évaluation si l'expression est bien parenthèsée. Ce point ne sera pas traité dans cet exercice.

(a) (1½ points) Voici par exemple, les états de la pile lors de l'exécution de l'algorithme pour $(3\times(4+6))$.



Dessiner les états de la pile lors de l'exécution de l'algorithme pour l'expression $((3\times 4)-6)$.

Solution:



(b) (4½ points) Pour cette question, vous pourrez utiliser les fonctions suivantes, de complexité $\Theta(1)$:

```
fonction estNombre(val v:objet): booleen; // Retourne Vrai si v est un nombre, Faux sinon
fonction estParentheseOuvrante(val v:objet){retourne v=='(';}
fonction estParentheseFermante(val v:objet){retourne v==')';}
fonction estOperateur(val v:objet){retourne v=='+' ou v=='-' ou v=='x';}
fonction estAddition(val v:objet){retourne v=='+';}
fonction estSoustraction(val v:objet){retourne v=='-';}
fonction estMultiplication(val v:objet){retourne v=='x';}
```

ainsi que les primitives liées à la structure de pile.

```
fonction valeur(val P:Pile d'objet): objet;
fonction pileVide(val P:Pile d'objet): booleen;
fonction creerPile(ref P:Pile d'objet): vide;
fonction empiler(ref P:Pile d'objet, val x:objet): vide;
fonction depiler (ref P:Pile d'objet): vide;
fonction detruirePile(ref P:Pile d'objet):vide;
```

Écrire une fonction `evaluer(val T:Tableau d'objet): entier` qui utilise une pile pour évaluer une expression arithmétique contenue dans un tableau. Vous ferez l'hypothèse que le tableau passé en paramètre contient une expression bien parenthèsée.

Solution:

```
fonction evaluer(val T:Tableau d'objet): entier;
// Hypothese : T est bien parenthesee
var P : Pile d'objet;
    op1, op2, res : entier;
    op : caractere;
debut
  creerPile(P);
  pour i de 0 a longueur(T)-1 faire {
    si estNombre(T[i]) ou estOperateur(T[i]) alors
      empiler(P,T[i]);
    sinon si estParentheseFermante(T[i]) alors {
      op2 = valeur(P);
      depiler(P);
      op = valeur(P);
      depiler(P);
      op1 = valeur(P);
      depiler(P);
      si estAddition(op) alors
        empiler(op1 + op2);
      sinon si estSoustraction(op) alors
        empiler(op1 - op2);
      sinon // op est une multiplication
        empiler(op1 * op2);
    }
  }
  res = valeur(P);
  depiler(P);
  retourner res;
fin
```

Une version **(non demandée)** qui contrôle le fait que l'expression est bien parenthèsée.

```
fonction evaluer(val T:Tableau d'objet): entier;
var P : Pile d'objet;
    op1, op2, res, cpt_po, cpt_pf, cpt_op : entier;
    op : caractere;
debut
  creerPile(P);
  cpt_po = 0;
  cpt_op = 0;
  cpt_pf = 0;
  pour i de 0 a longueur(T)-1 faire {
```

```
    si estParentheseOuvrante(T[i]) alors
      cpt_po += 1;
    sinon si estNombre(T[i]) alors
      empiler(P,T[i]);
    sinon si estOperateur(T[i]) alors {
      cpt_op += 1;
      empiler(P,T[i]);
    }
    sinon si estParentheseFermante(T[i]) alors {
      cpt_pf += 1;
      op2 = valeur(P);
      depiler(P);
      op = valeur(P);
      depiler(P);
      op1 = valeur(P);
      depiler(P);
      si estAddition(op) alors
        empiler(op1 + op2);
      sinon si estSoustraction(op) alors
        empiler(op1 - op2);
      sinon si estMultiplication(op) alors
        empiler(op1 * op2);
      sinon
        retourner NULL;
    }
    sinon // l'objet ne fait pas partie d'une expression bien parenthesee
      retourner NULL;
    // La partie prefixe d'une expression doit verifier la relation suivante
    si non ((cpt_po >= cpt_op) et (cpt_op  >= cpt_pf)) alors
      retourner NULL;
  }
  si cpt_po == cpt_pf et estNombre(valeur(P)) alors {
    res = valeur(P);
    depiler(P);
    si pileVide(P) alors
      retourner res;
  }
  detruirePile(P);
  retourner NULL;
fin
```

(c) (1 point) Donner et justifier la complexité de votre fonction `evaluer(val T:Tableau d'objet): entier`.

Solution: Une simple boucle `pour` qui n'utilise que des fonctions de complexité $\Theta(1)$. La complexité est donc en $\Theta(n)$ où n est la longueur du tableau, et donc de l'expression.

Exercice 3 : Les arbres rouge et noir (12 points)

Un arbre rouge et noir est un ABR comportant un bit de stockage supplémentaire par noeud : sa couleur, qui peut valoir soit `Rouge`, soit `Noir`. Cette couleur permet, comme le facteur d'équilibrage d'un AVL, que l'arbre soit *approximativemet* équilibré.

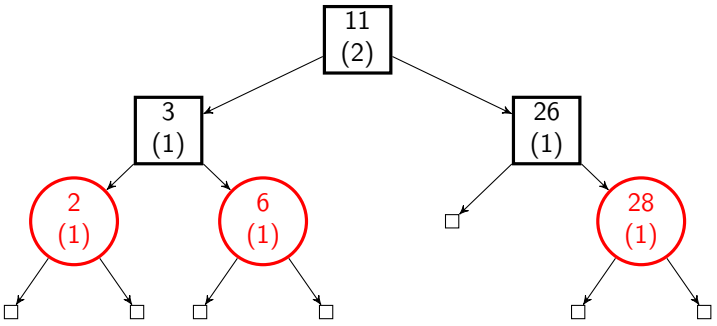
Définition : Un ABR est un **arbre rouge et noir** s'il satisfait les propriétés suivantes :

1. Chaque noeud est soit `Rouge`, soit `Noir`.
2. Chaque feuille `NIL` est `Noir`.
3. Si un noeud est `Rouge`, alors ses deux fils sont `Noir`.
4. Chaque chemin simple reliant un noeud q à une feuille descendante contient le même nombre de noeuds `Noir`. Ce nombre est appelé **hauteur noire** du noeud et est noté $hn(q)$.

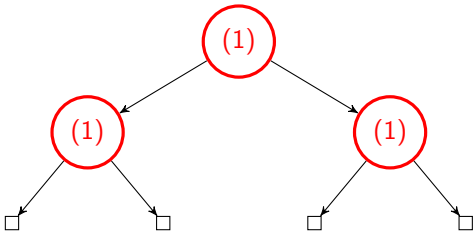
Par extension, pour un arbre rouge et noir A de racine r, la hauteur noire de A notée $hn(A)$ est égale à $hn(r)$.

Dans toutes les figures d'arbre rouge et noir, **les sommets `Noir` sont représentés par des carrés, les sommets `Rouge` par des cercles, et les sommets *indéterminés* par des carrés aux angles arrondis. Les sommets `Nil` de couleurs `Noir` sont représentés par des *petits* carrés.**

La figure suivante représente un arbre rouge et noir après l'insertion des valeurs `(6, 11, 26, 28, 2, 3)`. À chaque noeud, la hauteur noire est précisée entre parenthèses.

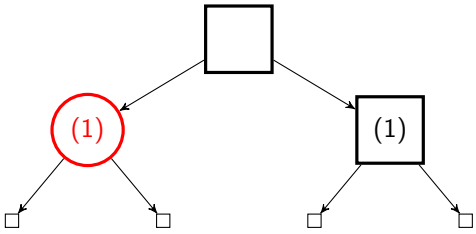


(a) i. (½ point) Pour chaque noeud q, précisez la valeur $hn(q)$ et justifiez le fait que l'ABR suivant n'est pas un arbre rouge et noir.



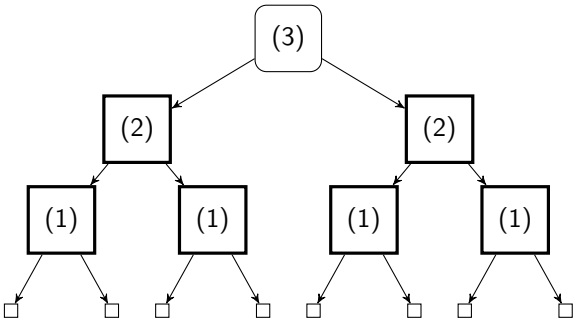
La racine étant `Rouge`, ses fils devraient être `Noir`.

ii. (½ point) Pour chaque noeud q, précisez la valeur $hn(q)$ et justifiez le fait que l'ABR suivant n'est pas un arbre rouge et noir.



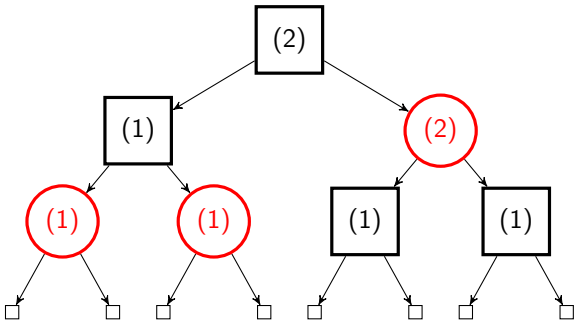
Les chemins depuis la racine contiennent 1 ou 2 noeuds `Noir`.

iii. (½ point) Colorez l'arbre suivant afin que la hauteur noire de la racine soit 3.



La racine est `Rouge` ou `Noir`.

iv. (½ point) Completez l'unique coloriage possible de l'arbre suivant en précisant les hauteurs noires de chaque noeud.



(b) L'objectif des questions suivantes est de montrer qu'un arbre rouge et noir est *approximativement équilibré*. Soit r la racine d'un arbre rouge et noir A, et $hn(A) = hn(r)$ la hauteur noire de A.

i. (½ point) Donnez un minorant de la hauteur de A notée $h(A)$ en fonction de $hn(A)$.

Solution: $h(A) \geq hn(A)$

ii. (½ point) Déduire de la propriété : "*Si un noeud est* `rouge`*, alors ses deux fils sont* `Noir`." un majorant de la hauteur de A notée $h(A)$ en fonction de $hn(A)$.

Solution: Entre deux noeuds `Noir`, il y a au plus un noeud `Rouge`. On en déduit : $h(A) \leq 2 \times hn(A)$

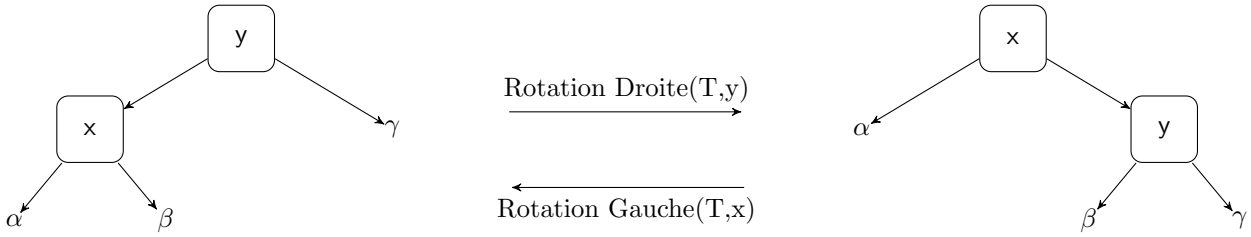
iii. (½ point) Rappel : Un arbre parfait complet de hauteur h contient $2^h - 1$ noeuds ; et pour tous les arbres binaires de n noeuds, $h \geq \log_2(n)$.
En supposant que l'arbre rouge et noir A contienne n noeuds, donnez un encadrement de $h(A)$ en fonction de n.

Solution: Pour tout arbre binaire A contenant n noeuds, on a $\log_2(n) \leq h(A)$.
Pour tout ARN A contenant n noeuds, on a $2^{hn(A)} \leq n \leq 2^{2 \times hn(A)}$.
On en déduit que pour tout ARN : $hn(A) \leq \log_2(n) \leq 2 \times hn(A)$.
puis que pour tout ARN : $hn(A) \leq \log_2(n) \leq 2 \times hn(A) \leq 2 \times \log_2(n)$.
Des deux questions précédentes, on déduit : $hn(A) \leq h(A) \leq 2 \times hn(A) \leq 2 \times \log_2(n)$.
Conclusion : $\log_2(n) \leq h(A) \leq 2 \times \log_2(n)$

iv. (½ point) Un arbre rouge et noir étant un ABR, la recherche d'un élément dans un arbre rouge et noir utilise l'algorithme de recherche dans un ABR. Donnez la complexité dans le pire des cas, de l'algorithme de reherche d'un élément dans un arbre rouge et noir A contenant n noeuds.

Solution: Le pire des cas correspond à une recherche infructueuse, car il faut parcourir une branche depuis la racine jusqu'à une feuille. Pour un ARN, il va donc falloir dans le pire des cas parcourir une branche de longueur $h(A)$, qui d'après la question précédente vérifie $\log_2(n) \leq h(A) \leq 2 \times \log_2(n)$. La complexité est donc en $\mathcal{O}(\ln(n))$ ou bien en $\mathcal{O}(\log_2(n))$. C'est équivalent.

(c) L'objectif des questions suivantes est d'écrire la fonction d'insertion dans un arbre rouge et noir. Comme pour les AVL, l'insertion dans un arbre rouge et noir se déroule en deux phases : la première insère le nouvel élément en utilisant la fonction d'insertion dans un ABR ; puis la seconde phase maintient les propriétés des couleurs des noeuds pour maintenir *approximativement* l'équilibre de l'arbre. Comme dans un AVL, pour maintenir les propriétés, l'algorithme *remonte* du sommet ajouté vers la racine. L'insertion dans un arbre rouge et noir utilise les mêmes fonctions de rotations droite et gauche que l'insertion dans un AVL.



Les questions précédentes ont montré que plusieurs coloriages peuvent être associés à un même arbre rouge et noir. Des choix sont donc nécessaire pour écrire la fonction d'insertion. Dans cet exercice, les voici :

1. La feuille ajoutée prend initialement la couleur `Rouge`.
2. La racine de l'arbre possède toujours la couleur `Noir`.

Dans la suite de l'exercice, vous supposerez que vous disposez de la structure suivante.

```
type celluleARN = structure {
  info : objet;
  gauche, droit, pere : sommet;
  couleur : mot; // "Rouge" ou "Noir";
  }
type sommet=^celluleARN;

fonction getCouleur(val s: sommet): mot;
debut
  si s == NIL alors
    retourne "Noir";
  sinon
    retourne s^.couleur;
fin

fonction setCouleur(ref s: sommet, val c:mot): vide;
debut
  s^.couleur = c;
fin
```

et de toutes les primitives des ABR (`filsDroit(s)`, `filsGauche(s)`, `pere(s)`, ...) pour pouvoir écrire la fonction d'insertion.

```
fonction ARN_Inserer(ref r: sommet, val v : objet): vide;
var s : sommet;
debut
  s = ABR_Inserer(r,v); // s est le sommet qui contient la nouvelle valeur "v"
  setCouleur(s,"Rouge");

          :

  setCouleur(r,"Noir"); // La racine est toujours coloriee en "Noir".
fin
```

Après avoir colorié `Rouge` le sommet ajouté, l'arbre vérifie les propriétés 1, 2 et 4 des arbres rouge et noir. Seule la propriété 3 "Si un noeud est `Rouge`, alors ses deux fils sont `Noir`." n'est peut-être pas vérifiée. L'algorithme va maintenir les propriétés vraies, et faire en sorte que la propriété 3 finisse par le devenir.

Nous allons distinguer quatre cas.

i. (1 point) Cas 0 : s est la racine ou le père du sommet s est de couleur `Noir`.

```
fonction estCas0(val s: sommet): boolean;
// retourne vrai s est la racine ou si le pere de s est 'Noir', faux sinon.
debut
  retourner s == r ou getCouleur(pere(s)) == "Noir";
fin
```

Expliquer pourquoi si `estCas0(s)` retourne `Vrai`, l'arbre est rouge et noir à la fin de l'algorithme.

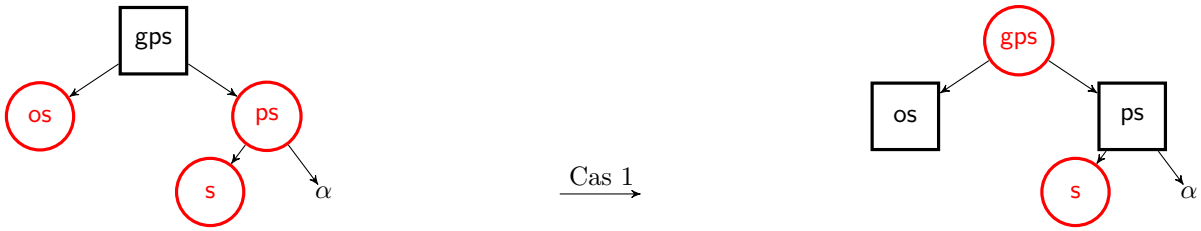
Solution: La contraposée de la propriété 3 est "Si au moins un fils est `Rouge`, alors le père est `Noir`. Le cas 0 satisfait donc la propriété 3 entre s et ps. L'arbre est donc un arbre rouge et noir.

Dans les dessins qui suivent, s de couleur `Rouge` est un sous arbre rouge et noir, qui contient le sommet ajouté (s est soit le sommet ajouté, soit un de ses ancètres lors de l'itération du processus), ps est le père de s, gps est le grand-père de s et os l'oncle de s.

Seule la propriété 3 entre s et ps n'est pas vérifiée.

ii. (1½ points) Cas 1 : Le père du sommet s ajouté est de couleur `Rouge`, et l'oncle (qui existe car le père `Rouge` ne peut pas être la racine qui est `Noir`) du sommet ajouté est de couleur `Rouge`.

Le père de s ne peut pas rester `rouge`. Une solution consiste à appliquer la transformation suivante qui repousse le problème vers la racine.



Sans oublier les cas symétriques du cas dessiné, complétez le code des deux fonctions suivantes :

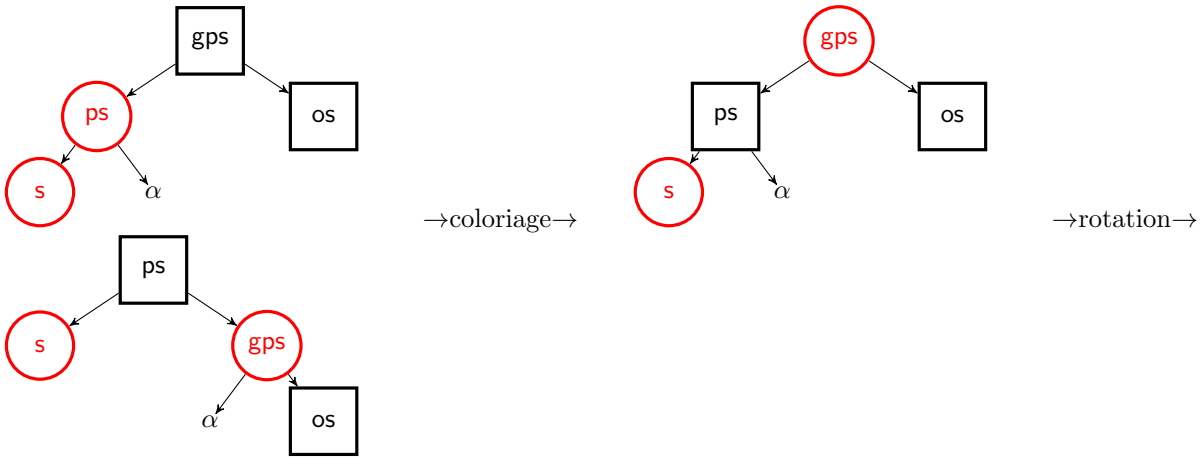
Solution:

```
fonction estCas1(val s: sommet): boolean;
// retourne vrai si l'oncle de s est 'Rouge', faux sinon.
debut
// on sait que pere(s) est 'Rouge'.
// l'oncle est 'Rouge' si les deux freres sont de la meme couleur
  retourner getCouleur(filsDroit(pere(pere(s)))) ==
            getCouleur(filsGauche(pere(pere(s))));
fin

fonction traiterCas1(val s: sommet): sommet;
// Colorie pere, oncle et grand-pere, puis retourne le grand-pere
debut
  setCouleur(pere(pere(s)),'Rouge');
  setCouleur(filsGauche(pere(pere(s))),'Noir');
  setCouleur(filsDroit(pere(pere(s))),'Noir');
  retourner pere(pere(s));
fin
```

iii. (1½ points) Cas 2 : Le père du sommet s ajouté est de couleur `Rouge`, et l'oncle (qui existe car le père `Rouge` ne peut pas être la racine qui est `Noir`) du sommet ajouté est de couleur `Noir`. De plus le lien (fils gauche ou droit) entre s et $pere(s)$ est le même que celui entre $pere(s)$ et $pere(pere(s))$.

Le père de s ne peut pas rester `rouge`. Une solution consiste à appliquer la transformation suivante :



Sans oublier les cas symétriques du cas dessiné, complétez le code des deux fonctions suivantes qui résoud le problème de la propriété 3.

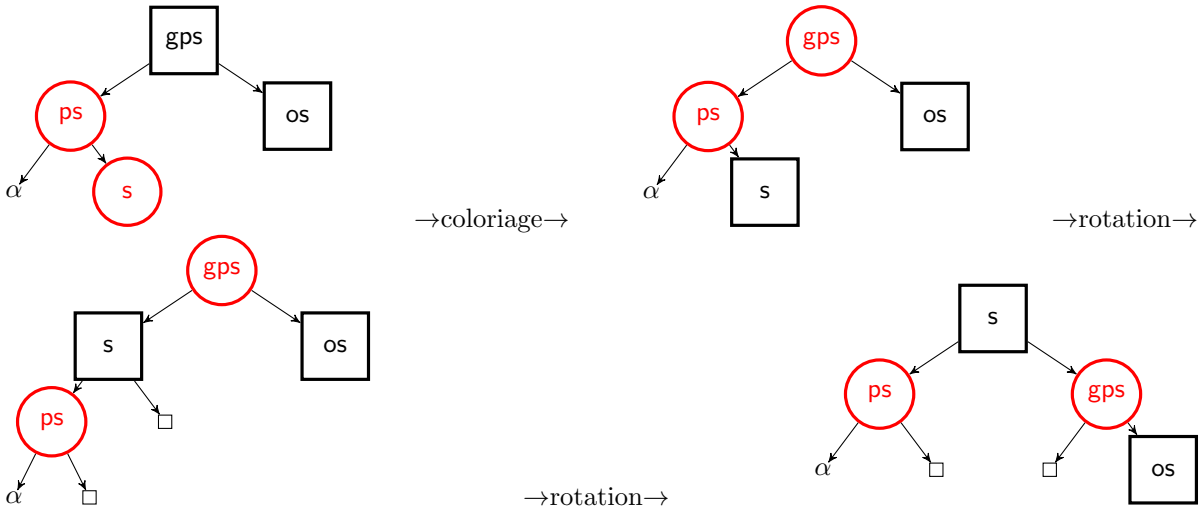
Solution:

```
fonction estCas2(val s: sommet): boolean;
// retourne vrai si l'oncle de s est 'Noir' et
//   si les relations (s,pere(s)) et (pere(s),pere(pere(s))) sont identiques,
// faux sinon.
debut
// on sait que pere(s) est 'Rouge'.
// l'oncle est 'Noir' si les deux freres ne sont pas de la meme couleur
  retourner (non (getCouleur(filsDroit(pere(pere(s)))) ==
                  getCouleur(filsGauche(pere(pere(s))))))
            et
            (s == filsDroit(filsDroit(pere(pere(s)))) ou
             s == filsGauche(filsGauche(pere(pere(s)))));
fin

fonction traiterCas2(val s: sommet): sommet;
// Colorie pere et grand-pere, une rotation puis retourne s
debut
  setCouleur(pere(s),'Noir');
  setCouleur(pere(pere(s)),'Rouge');
  si s == filsGauche(pere(s)) alors
    rotationDroite(pere(pere(s)));
  sinon
    rotationGauche(pere(pere(s)));
  retourner s;
fin
```

iv. (1½ points) Cas 3 : Le père du sommet s ajouté est de couleur `Rouge`, et l'oncle (qui existe car le père `Rouge` ne peut pas être la racine qui est `Noir`) du sommet ajouté est de couleur `Noir`. De plus le lien (fils gauche ou droit) entre s et $pere(s)$ est différent de celui entre $pere(s)$ et $pere(pere(s))$.

Le père de s ne peut pas rester `rouge`. Une solution consiste à appliquer la transformation suivante :



Sans oublier les cas symétriques du cas dessiné, complétez le code des deux fonctions suivantes qui résoud le problème de la propriété 3.

Solution:

```
fonction estCas3(val s: sommet): boolean;
// retourne vrai si l'oncle de s est 'Noir' et
//   si les relations (s,pere(s)) et (pere(s),pere(pere(s))) sont differentes,
// faux sinon.
debut
// on sait que pere(s) est 'Rouge'.
// l'oncle est 'Noir' si les deux freres ne sont pas de la meme couleur
  retourner (non (getCouleur(filsDroit(pere(pere(s)))) ==
                  getCouleur(filsGauche(pere(pere(s))))))
            et
            (non (s == filsDroit(filsDroit(pere(pere(s)))) ou
                  s == filsGauche(filsGauche(pere(pere(s))))));
fin

fonction traiterCas3(val s: sommet): sommet;
// Colorie s et grand-pere, deux rotations puis retourne le pere de s
var ps : sommet;
debut
  ps = pere(s);
  pps = pere(ps);
  setCouleur(s,'Noir');
  setCouleur(pps,'Rouge');
  si s == filsDroit(ps) alors {
    rotationGauche(ps);
    rotationDroite(pps);
  }
  sinon {
    rotationDroite(ps);
    rotationGauche(pps);
  }
  retourner ps;
fin
```

v. ($1\frac{1}{2}$ points) Itération du processus : Les cas 2 et 3 ne nécessitent pas de continuer la remontée vers la racine. Le cas 1 qui change la couleur du grand-père de s peut engendrer d'autres modifications.
Le critère d'arrêt de l'itération correspond soit à la racine, soit à un père `Noir`, c'est à dire au cas 0.
Complétez le code de la fonction suivante :

Solution:

```
fonction ARN_Inserer(ref r: sommet, val v : objet): vide;
var s : sommet;
debut
  s = ABR_Inserer(r,v); // s est le sommet qui contient la nouvelle valeur "v"
  setCouleur(s,"Rouge");
  tant que non estCas0(s) faire {
    si estCas1(s) alors
      s = traiterCas1(s);
    sinon si estCas2(s) alors
      s = traiterCas2(s);
    sinon // c'est le cas 3
      s = traiterCas3(s);
    }
  setCouleur(r,"Noir"); // La racine est toujours coloriee en "Noir".
fin
```

vi. (1 point) Donnez la complexité dans le pire des cas de votre fonction `ARN_insérer`.

Solution: $\mathcal{O}(\ln(n))$ ou bien $\mathcal{O}(\log_2(n))$. C'est équivalent.