

Indiquez votre code d'anonymat :

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 $L = (6, 11, 26, 28, 2, 3)$.

Étape 1	Étape 2	Étape 3
Étape 4	Étape 5	Étape 6

(b) ($1\frac{1}{2}$ points) Tas min : insertion de $L = (6, 11, 26, 28, 2, 3)$.

Étape 1	Étape 2	Étape 3
Étape 4	Étape 5	Étape 6

(c) ($1\frac{1}{2}$ points) AVL : insertion de $L = (6, 11, 26, 28, 2, 3)$.

Étape 1	Étape 2	Étape 3
Étape 4	Étape 5	Étape 6

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) × (-3 + 6)) - 5)	Vrai	55	
(356)	Faux	-	Trop parenthésée
5 + 15	Faux	-	Pas suffisamment parenthésée
(5 + 15 + 10)	Faux	-	Pas suffisamment parenthésée
(5 + 15 × 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) × (-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 :

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

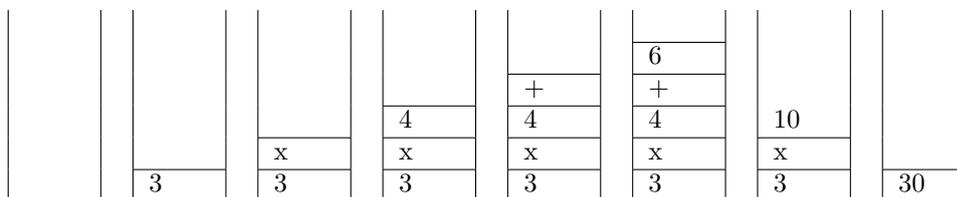
avec E1 et E2 deux expressions bien parenthésées et "op" un opérateur arithmétique binaire.

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 parenthésée. 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 × (4 + 6)).



(b) ($4\frac{1}{2}$ points) Pour cette question, vous pourrez utiliser les fonctions suivantes, de complexité $\Theta(1)$:

```
fonction estNombre(val v:objet): boolean; // 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): boolean;
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 parenthésée.

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

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 *approximativement* équilibré.

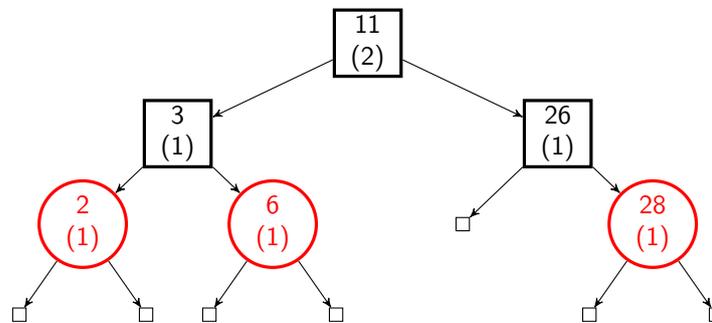
Définition : Un ABR est un **arbre rouge et noir** s'il satisfait les propriétés suivantes :

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 $hn(q)$.

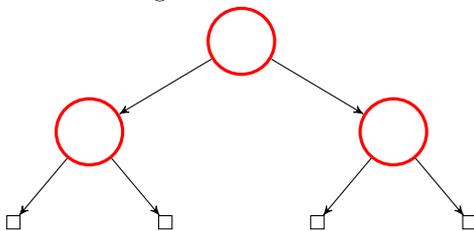
Par extension, pour un arbre rouge et noir A de racine r , la hauteur noire de A notée $hn(A)$ est égale à $hn(r)$.

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*.

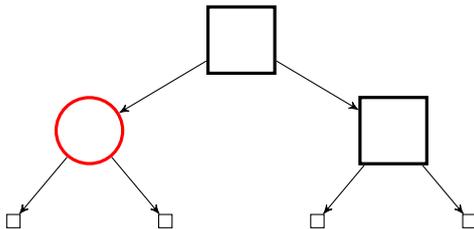
La figure suivante représente un arbre rouge et noir après l'insertion des valeurs (6, 11, 26, 28, 2, 3). À chaque noeud, la hauteur noire est précisée entre parenthèses.



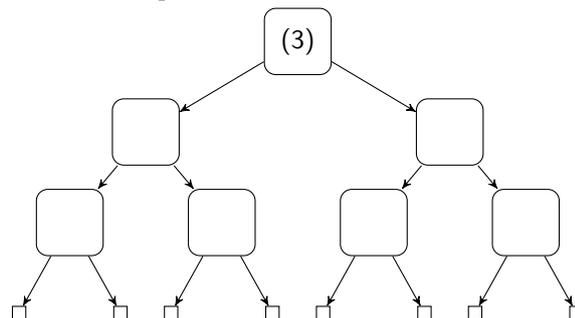
- (a) i. ($1/2$ point) Pour chaque noeud q , précisez la valeur $hn(q)$ et justifiez le fait que l'ABR suivant n'est pas un arbre rouge et noir.



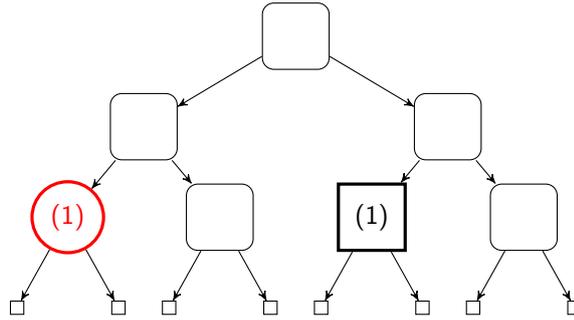
- ii. ($1/2$ point) Pour chaque noeud q , précisez la valeur $hn(q)$ et justifiez le fait que l'ABR suivant n'est pas un arbre rouge et noir.



- iii. ($1/2$ point) Colorez l'arbre suivant afin que la hauteur noire de la racine soit 3.



- iv. ($\frac{1}{2}$ 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. ($\frac{1}{2}$ point) Donnez un minorant de la hauteur de A notée $h(A)$ en fonction de $hn(A)$.

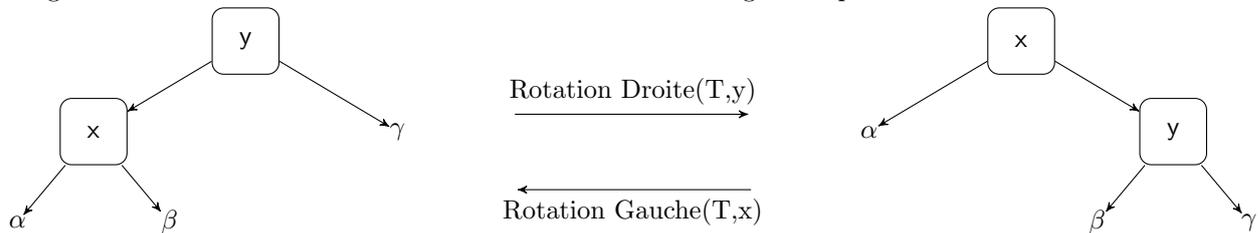
- ii. ($\frac{1}{2}$ point) Déduire de la propriété : "Si un noeud est rouge, alors ses deux fils sont Noir." un majorant de la hauteur de A notée $h(A)$ en fonction de $hn(A)$.

- iii. ($\frac{1}{2}$ 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 .

- iv. ($\frac{1}{2}$ 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 recherche d'un élément dans un arbre rouge et noir A contenant n noeuds.

- (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 nécessaires 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

```

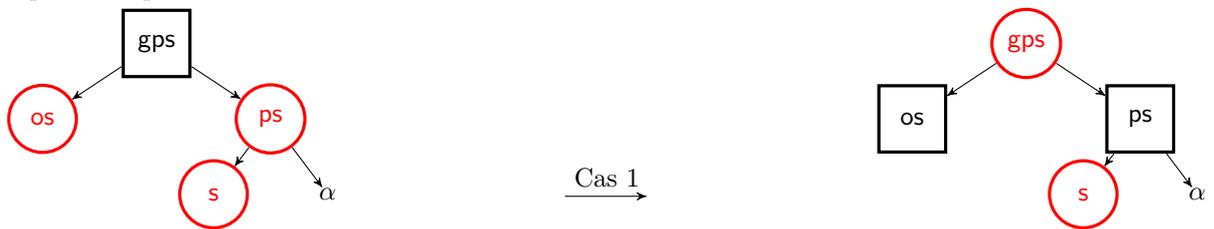
Expliquer pourquoi si `estCas0(s)` retourne **Vrai**, l'arbre est rouge et noir à la fin de l'algorithme.

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 ancêtres 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.



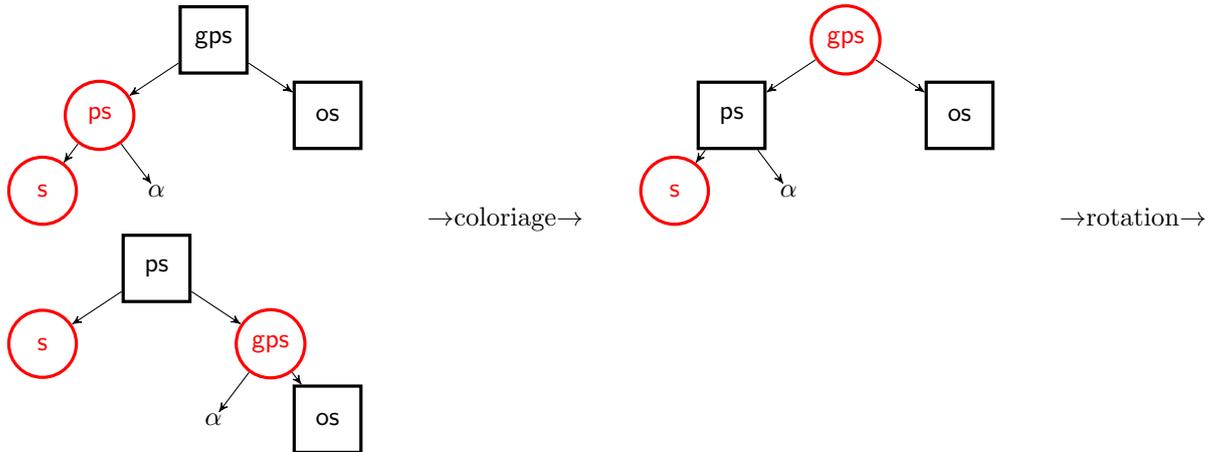
Sans oublier les cas symétriques du cas dessiné, complétez le code des deux fonctions suivantes :

```
fonction estCas1(val s: sommet): boolean;
// retourne vrai si l'oncle de s est 'Rouge', faux sinon.
```

```
fonction traiterCas1(val s: sommet): sommet;
// Colorie pere, oncle et grand-pere, puis retourne le grand-pere
```

- 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.

```

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.

```

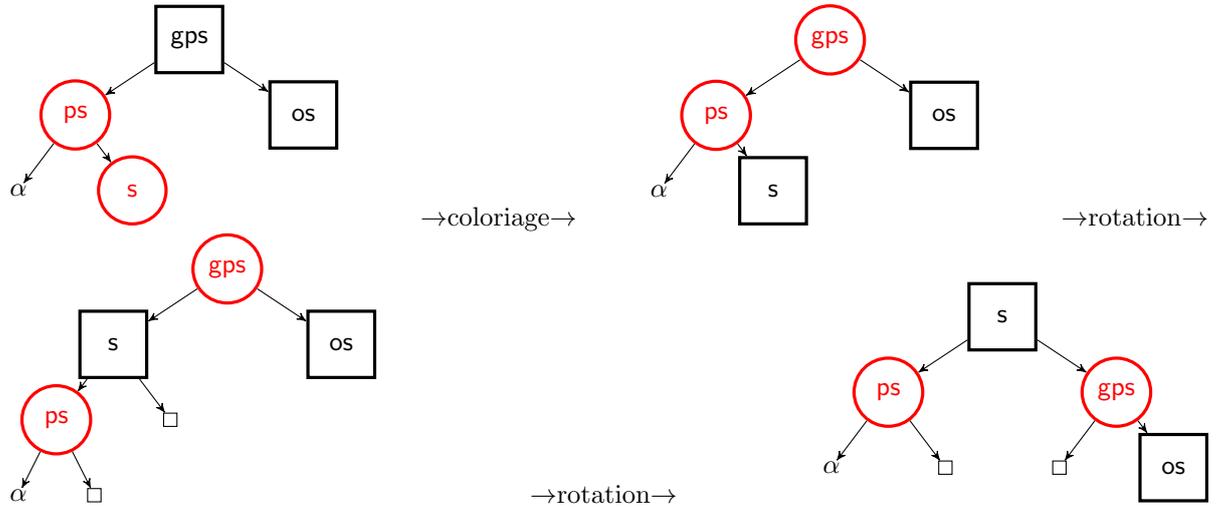
```

fonction traiterCas2(val s: sommet): sommet;
// Colorie pere et grand-pere, une rotation puis retourne s

```

- 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.

```

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.

```

```

fonction traiterCas3(val s: sommet): sommet;
// Colorie s et grand-pere, deux rotations puis retourne le pere de s

```

- v. (1½ 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 :

```
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
```

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