Les arbres bicolores (rouge-noir)
Correction

1 De l'arbre 2-3-4 à l'arbre bicolore

Solution 1.1 **(Propriétés)**

1. Un *arbre bicolore* est un arbre binaire de recherche dont les nœuds portent une information supplémentaire : ils sont rouges ou noirs (ou blancs !). C'est une représentation des arbres 2-3-4.

2. Chaque nœud de l'arbre 2-3-4 est représenté par un nœud noir contenant une des clés du nœud arbre 2-3-4 et les autres clés sont dans des nœuds rouges fils du nœud noir :

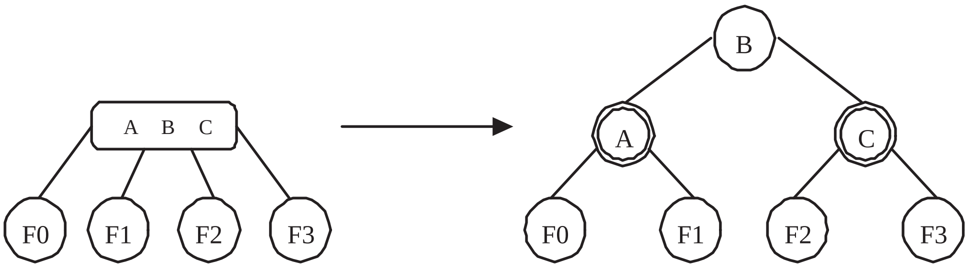

Fig. 1 – Transformation d'un 4-nœud

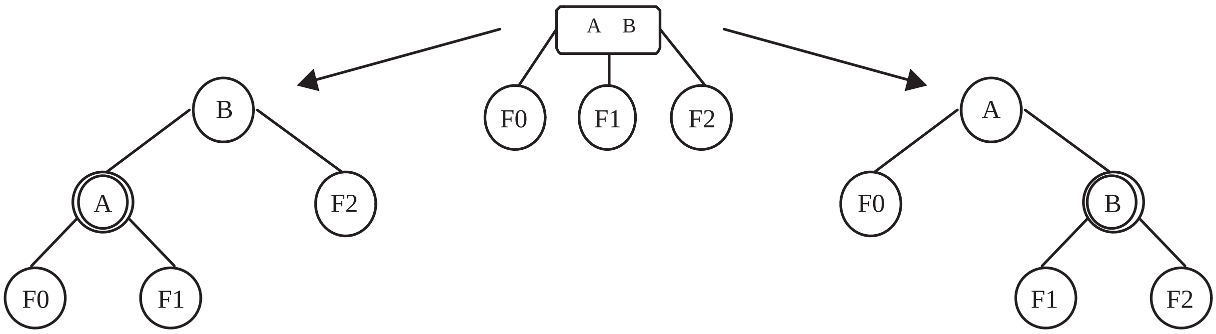

Fig. 2 – Transformation d'un 3-nœud

Un 2-nœud devient un nœud noir..

3. – C'est un arbre binaire de recherche.
   – Un nœud rouge ne peut pas avoir de fils rouge.
   – La couleur de la racine est toujours noire.
   – Le nombre de nœuds noirs sur tous les chemins de la racine aux feuilles est le même (hauteur noire = hauteur arbre 2-3-4).

Solution 1.2 **(Arbres bicolores et 2-3-4 : Mesures)**

1. – La taille d'un arbre 2-3-4 est représentée par le nombre de nœuds noirs de l'arbre bicolore correspondant.
   – Sa hauteur est les nombre de nœuds noirs sur n'importe quel chemin de la racine à une feuille (sans compter la racine).

2. **Spécifications :**

   La procédure `hauteur_taille (t_arn` A`, entier` t`,` h`)` affecte aux variables t et h la taille et la hauteur de l'arbre 2.3.4 représenté par l'arbre bicolore A.

   Principe :

   Si l'arbre est vide, la taille est 0, la hauteur -1.
   Sinon, on récupère les tailles et hauteurs des fils gauche et droit.
   La taille est la somme des tailles des fils gauche et droit, plus un si le nœud actuel est noir.
   La hauteur est celle d'un des deux fils (identiques de toute façon), incrémentée si le nœud actuel est noir.

```
algorithme procedure hauteur_taille
    parametres locaux
        t_arn    A
    parametres globaux
        entier    taille, hauteur

    variables
        entier    tg, td
debut
    si A = NUL alors
        taille ← 0
        hauteur ← -1
    sinon
        hauteur_taille (A↑.fg, tg, hauteur)
        hauteur_taille (A↑.fd, td, hauteur)
        si A↑.rouge alors
            taille ← tg + td
        sinon
            taille ← tg + td + 1
            hauteur ← hauteur + 1
        fin si
fin algorithme procedure hauteur_taille
```

Solution 1.3 **(De l'arbre 2-3-4 à l'arbre bicolore)**

1.

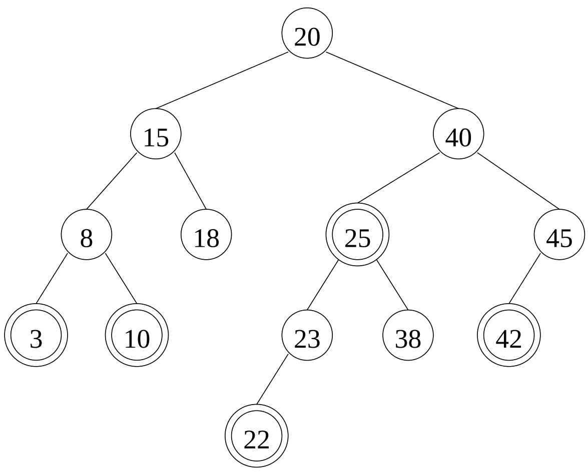

FIG. 3 – Arbre bicolore

2. **Spécifications :**

   La fonction `noeud_bic (t_element` *cle*`, booleen` *rouge*`, t_arn` *fg*`,` *fd*`) : t_arn` retourne l'arbre bicolore dont tous les champs sont donnés en paramètres.

```
algorithme fonction noeud_bic : t_arn
    parametres locaux
        t_element    cle
        booleen      rouge
        t_arn        fg, fd

    variables
        t_arn    B
debut
    allouer (B)
    B↑.cle ← cle
    B↑.rouge ← rouge
    B↑.fg ← fg
    B↑.fd ← fd
    retourne (B)
fin algorithme fonction noeud_bic
```

Spécifications :

La fonction `transf (t_a234` *A*`) : t_arn` retourne l'arbre bicolore correspondant à l'arbre 2-3-4 *A*.

```
algorithme fonction transf : t_arn
    parametres locaux
        t_a234     A

    variables
        t_arn      P, T
debut
    si A = NUL alors
        retourne (NUL)
    sinon
        P ← noeud_bic (A↑.cle[1], faux, transf (A↑.fils[1]), transf (A↑.fils[2]))
        si A↑.nbcles = 1 alors
            retourne (P)
        sinon
            P↑.rouge ← vrai
            T ← noeud_bic (A↑.cle[2], faux, P, NUL)
            si A↑.nbcles = 2 alors
                T↑.fd ← a234_to_bic (A↑.fils[3])
            sinon
                T↑.fd ← noeud_bic (A↑.cle[3], vrai, a234_to_bic (A↑.fils[3]),
                                                    a234_to_bic (A↑.fils[4]))
            fin si
            retourne (T)
        fin si
    fin si
fin algorithme fonction transf
```

3. **Spécifications :**
   La fonction ARNtoA234 (t_arn A) : t_a234 retourne l'arbre 2-3-4 correspondant à l'arbre bicolore A.

La fonction construit à chaque appel un nœud de l'arbre 2.3.4. complet. Chaque nœud noir est traité avec ses fils rouges, s'ils existent, pour construire le nœud 2.3.4. correspondant. Les fils étant créés de manière récursive, à chaque appel de la fonction, la racine de l'arbre bicolore à transformer est donc noire !

```
algorithme fonction ARNtoA234 : a234
   parametres locaux
      t_arn    A

   variables
      t_a234   B
      entier i
debut
    si A = NUL alors
        retourne (NUL)
    sinon
             /* le nœ ud courant est toujours un nœ ud noir */
        allouer(B)
                                        /* Le fils gauche */
        si (A↑.fg <> NUL) et A↑.fg↑.rouge alors
            B↑.cle[1] ← A↑.fg↑.cle                  /* ∃ et est rouge */
            B↑.fils[1] ← ARNtoA234(A↑.fg↑.fg)
            B↑.fils[2] ← ARNtoA234(A↑.fg↑.fd)
            B↑.cle[2] ← A↑.cle
            i ← 3
        sinon
            B↑.cle[1] ← A↑.cle                      /* ∄ ou est noir */
            B↑.fils[1] ← ARNtoA234(A↑.fg)
            i ← 2
        fin si
                                        /* Le fils droit */
        si (A↑.fd <> NUL) et A↑.fd↑.rouge alors
            B↑.cle[i] ← A↑.fd↑.cle                  /* ∃ et est rouge */
            B↑.fils[i] ← ARNtoA234(A↑.fd↑.fg)
            B↑.fils[i+1] ← ARNtoA234(A↑.fd↑.fd)
            B↑.nbcles ← i
        sinon
            B↑.fils[i] ← ARNtoA234(A↑.fd)           /* ∄ ou est noir */
            B↑.nbcles ← i-1
        fin si

        retourne (B)
    fin si

fin algorithme fonction ARNtoA234
```

2 Modifications

Solution 2.1 **(Insertions)**

1. (a) *L'éclatement :*

```
algorithme procedure eclate
    parametres locaux
        t_arn   A
debut
    A↑.rouge ← Vrai
    A↑.fg↑.rouge ← Faux
    A↑.fd↑.rouge ← Faux
fin algorithme procedure eclate
```

(b) *Les rotations :*

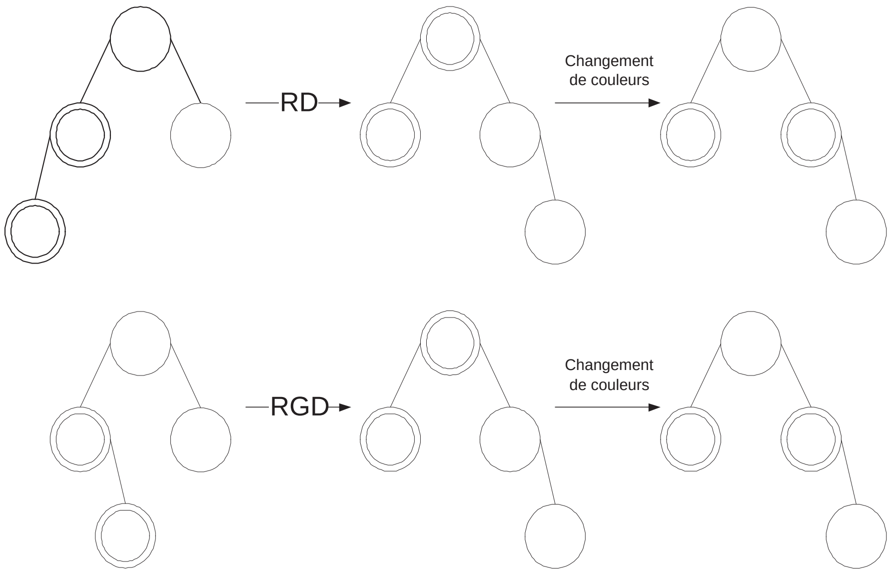

FIG. 4 – Exemples de rotation-bicolores sur arbre bicolore.

```
algorithme procedure RD
    parametres globaux
        t_arn   A

    variables
        t_arn   FG
debut
    FG ← A↑.fg
    A↑.fg ← FG↑.fd
    FG↑.fd ← A
    A↑.rouge ← Vrai
    FG↑.rouge ← Faux
    A ← FG
fin algorithme procedure RD
```

```
algorithme procedure RGD
    parametres globaux
        t_arn   A

    variables
        t_arn   FGD
debut
      FGD ← A↑.fg↑.fd
      A↑.fg↑.fd ← FGD↑.fg
      FGD↑.fg ← A↑.fg
      A↑.fg ← FGD↑.fd
      FGD↑.fd ← A
      A↑.rouge ← Vrai
      FGD↑.rouge ← Faux
      A ← FGD
algorithme procedure RGD
```

2. L'algorithme est récursif. La descente se fait selon le principe classique d'insertion aux feuilles dans un arbre binaire de recherche.

   En remontant, l'algorithme retourne le nombre de nœuds rouges consécutifs rencontrés (si deux noeuds rouges sont consécutifs, l'un est le "fils rouge" et l'autre le "petit-fils rouge", les transformations se faisant sur le "père noir"). Ce nombre est signé : -2 indique que le "petit-fils" rouge est à droite, +2 qu'il est à gauche.
   - Si le fils rouge possède un frère rouge, on éclate.
   - Si le frère du fils rouge est noir (ou n'existe pas), on effectue une rotation.

3. *Note :* Lorsque la valeur de retour de la fonction est 0, on peut tout simplement retourner 0 quelque soit la couleur de la racine actuelle. En effet, il n'y aura plus aucune transformation à faire !

   La procédure d'appel : en revenant de l'insertion la racine doit être noire !

```
algorithme procedure insertARN
    parametres locaux
        t_element   x
    parametres globaux
        t_arn       A

debut
   si InsertionARN (x, A) = 1 alors
         /* La racine est rouge à la suite d'un éclatement, ou si elle était vide */
         A↑.rouge ← faux
   fin si
fin algorithme procedure insertARN
```

```
algorithme fonction InsertionARN : entier
    parametres locaux
        t_element    x
    parametres globaux
        t_arn        A

    variables
        entier   nr      /* Nombres de nœuds rouges consécutifs */
debut
     si A = NUL alors
         A ← noeud_bic (x, vrai, NUL, NUL)
         retourne (1)
     fin si
     si x = A↑.cle alors
         retourne (0)
     fin si
     si x < A↑.cle alors
        nr ← InsertionARN (x, A↑.fg)
        si A↑.rouge alors        /* noeud rouge, retourne 1 ou 2 */
           retourne (1+nr)
        sinon                   /* noeud noir */
           si abs(nr) = 2 alors
              si (A↑.fd <> NUL) et A↑.fd↑.rouge alors   /* éclatement */
                 eclate(A)
                 retourne (1)
              sinon                        /* rotation-bicolore */
                 si nr = 2 alors
                    RD(A)
                 sinon
                    RGD(A)
                 fin si

              fin si
           fin si
           retourne (0)
        fin si
     sinon      /* x > A↑.cle */
        nr ← InsertionARN (x, A↑.fd)
        si A↑.rouge alors        /* noeud rouge, retourne 1 ou -2 */
            retourne (1-3*nr)
        sinon                   /* noeud noir */
            si abs(nr) = 2 alors
               si (A↑.fg <> NUL) et A↑.fg↑.rouge alors   /* éclatement */
                  eclate(A)
                  retourne (1)
               sinon                        /* rotation-bicolore */
                  si nr = -2 alors
                      RG(A)
                  sinon
                      RDG(A)
                  fin si
               fin si
            fin si
            retourne (0)
        fin si
    fin si
fin algorithme fonction InsertionARN
```

Solution 2.2 **(Suppression ?)**

1. *Quel problème peut poser la suppression dans un arbre bicolore ? Quel est le lien avec la suppression dans un arbre 2.3.4. ?*

   Si on supprime un nœud noir, le nombre de nœuds noirs sur tous les chemins n'est plus le même. Ce qui revient à vouloir supprimer un 2-nœud dans un arbre 2.3.4.

2. *Supprimer la clé 3 dans l'arbre de la figure 3, proposer une solution pour réparer l'arbre. Comparer avec la transformation envisagée dans l'arbre 2.3.4. correspondant. Généraliser la transformation.*

   Si on supprime la clé 3, en inversant la couleur entre le père et le frère (figure 5) de la clé 3, on rétablit le nombre de nœuds noirs sur tous les chemins (figure 6). Cela s'apparente à une fusion dans l'arbre 2.3.4..

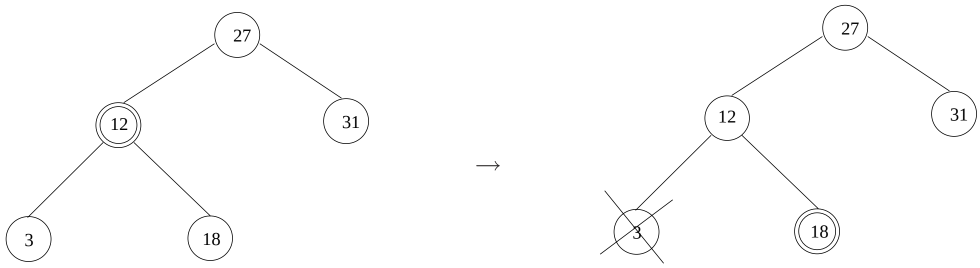

FIG. 5 – Avant   FIG. 6 – Après

   Pour généraliser (figure 7) : il suffit de remarquer que le frère devient toujours rouge et que le père devient noir même s'il l'était déjà avant. Mais dans ce cas le nombre de nœuds noirs n'est pas rétabli et il va falloir remonter dans l'arbre pour "réparer".

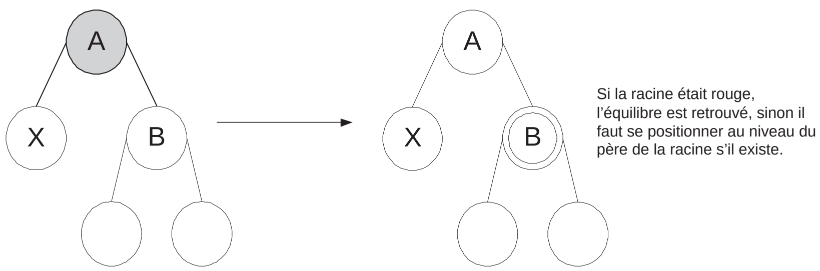

FIG. 7 – **Cas n° F** - Réparation après suppression dans le sous-arbre gauche

3. *Supprimer la clé 3 dans l'arbre de la figure 4, proposer une solution pour réparer l'arbre. Comparer avec la transformation envisagée dans l'arbre 2.3.4. correspondant. Généraliser la transformation.*

   Si on supprime la clé 3, en faisant une rotation su r 12 (figure 8, ce n'est pas une rotation bicolore), 18 devenant rouge, et 12 et 21 noirs, le nombre de nœuds noirs est rétabli (figure 9). Cela s'apparente à une rotation dans l'arbre 2.3.4..

   Pour généraliser (figure 10) : malgré la rotation, la racine conserve sa couleur et ses deux fils deviennent noirs. L'arbre est rééquilibré en terme de nœuds noirs.

4. *Supprimer la clé 3 dans l'arbre de la figure 5, quelle transformation peut-on envisager pour se retrouver dans le cas précédent ? Généraliser.*

   Une rotation-bicolore droite du sous-arbre droit (figure 12) et nous sommes revenu au cas précédent (figure 13).

   Pour généraliser (figure 14) :

5. *Supprimer la clé 3 dans l'arbre de la figure 6, proposer une solution pour se ramener à un des trois cas précédents. Généraliser.*

   Une rotation-bicolore gauche de l'arbre (figure 16). Nous sommes dans le cas n° F (figure 17).

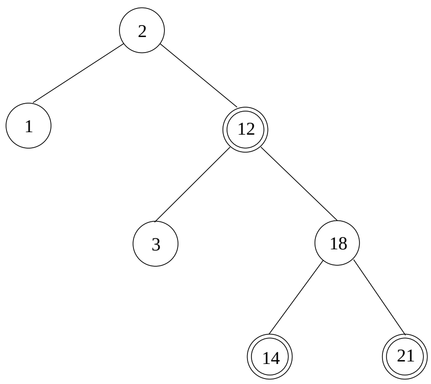

Fig. 8 – Avant

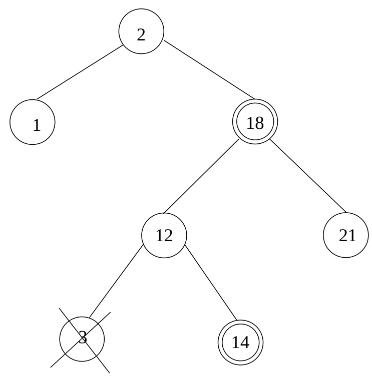

Fig. 9 – Après

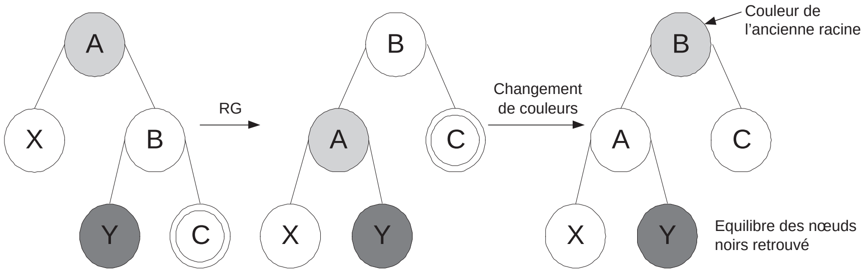

Fig. 10 – **Cas n° R** - Réparation après suppression dans le sous-arbre gauche

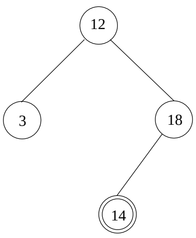

Fig. 11 – Avant

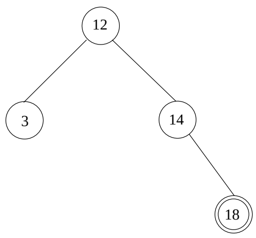

Fig. 12 – Pendant

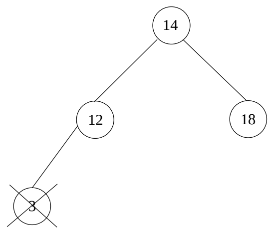

Fig. 13 – Après

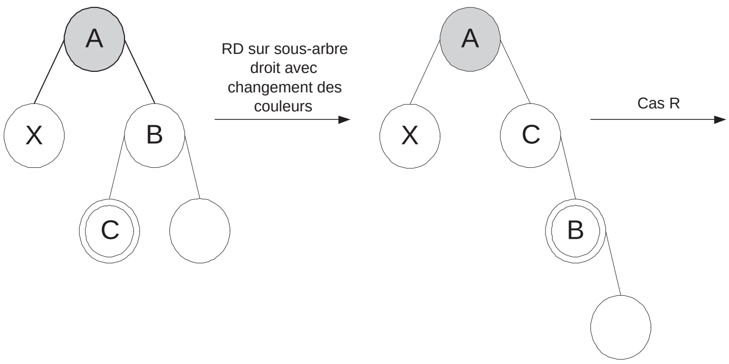

Fig. 14 – **Cas n° M1** - Modification avant réparation (par le cas n° R), après suppression dans le sous-arbre gauche

Pour généraliser (figure 18) : Le cas n° M2 est toujours suivi d'un des autres cas. Si la réparation envisagée est le cas n° F, sa racine est toujours rouge, donc dans tous les cas, l'arbre est rééquilibré.

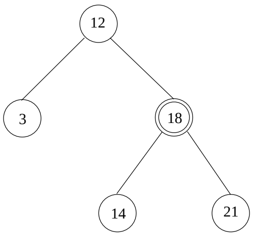

Fig. 15 – Avant

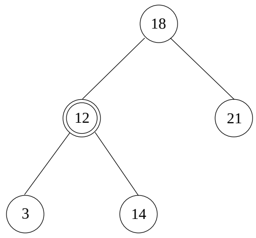

Fig. 16 – Pendant

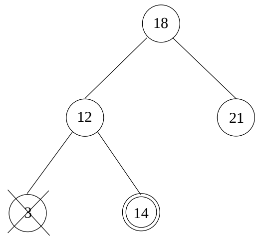

Fig. 17 – Après

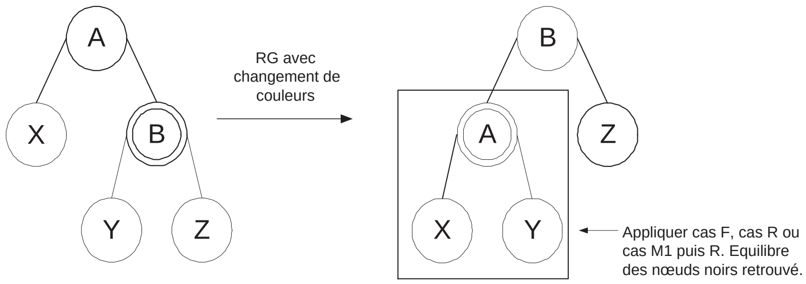

Fig. 18 – **Cas n° M2** - Modifiation avant réparation possible (avec Cas n° F ou Cas n° R ou Cas n° M1 et R), après suppression dans le sous-arbre gauche

6. *En déduire le principe de suppression dans un arbre bicolore.*
   La suppression dans un arbre bicolore correspond à la suppression dans un arbre 2.3.4. à la remontée.

   - A la descente : recherche de la clé
     - Si c'est une feuille, on la supprime directement.
     - Si c'est un nœud interne, sa clé est remplacée par le maximum du sous-arbre gauche s'il existe. Il faut ensuite supprimer le maximum du fils gauche. Donc, la suppression physique se fait en feuille.
     - Si c'est un nœud simple à droite, alors le nœud est remplacé par son fils droit qui est forcément rouge (et qui devient donc noir).
   - A la remontée : Si suppression d'un nœud noir alors il faut "réparer" l'arbre.

     Les questions 2 à 5, présentent les différents cas qui peuvent être rencontrés lors d'une suppression effectuée dans le sous-arbre gauche : chaque "cas" indique la transformation à effectuer pour réparer l'arbre.
     Dans les figures 7 à 10, x représente :
     - soit la feuille à supprimer,
     - soit la racine du sous-arbre dans lequel un nœud noir a été supprimé et où il subsiste un déséquilibre des nœuds noirs (correspond à un nœud vide dans l'arbre 2.3.4.).