Arbres binaires de recherche équilibrés - Arbres Rouge-Noir.

Structures de données - Travaux dirigés sur machines

Séance 11 et 12

Les travaux seront réalisés à partir de l'archive `SD-TP10.tgz` fournie sur moodle et contenant la hiérarchie suivante :

`SD-TP10`
–> `Test` : répertoire contenant les fichiers de test.
–> `Code` : répertoire contenant le code source fourni (programme principal et interface des arbres binaire de recherche uniquement)

Pour réaliser ce TP, les étudiant devront repartir de leur module de gestion des arbres binaire de recherche développé au TP précédent.
Pour cela, **recopier** votre fichier `bstree.c` dans le répertoire `Code`. L'interface `bstree.h` n'ayant pas due être modifiée, vous devez conserver celle fournie dans l'archive de ce TP.
À la demande de l'enseignant, l'étudiant devra pouvoir fournir une archive similaire à l'archive de départ contenant le résultat de son travail.

Table des matières

1 Description de l'archive logicielle fournie **2**

2 Propriétés des arbres Rouge-Noir **3**

3 Coloration et rotations. **4**
3.1 Exercice 1 : extension de la structure de données et coloration de l'arbre. 4
3.2 Exercice 2 : programmation des opérateurs de rotation. 5

4 Insertion d'une valeur dans un arbre rouge-noir. **7**
4.1 Restauration des propriétés des arbres Rouge-Noir après insertion. 7
4.2 Exercice 3 : programmation des opérateurs de correction après insertion. 9
4.3 Exercice 4 : recherche de valeurs dans un arbre Rouge-Noir. 10

5 Suppression d'une valeur dans un arbre rouge-noir. **11**
5.1 Restauration des propriétés des arbres Rouge-Noir après insertion. 12
5.2 Exercice 5 : programmation des opérateurs de correction après suppression. . . . 14

L'objectif de ce TP, qui court sur 2 séances, est de programmer le module de gestion des arbres Rouge-Noir, arbres binaires de recherche équilibrés, et d'en effectuer la visualisation en utilisant

l'outil `dot` de la suite logicielle graphviz [1] comme montré sur la figure 1. Issu du fichier de test `Test/testfile2.txt`, cet arbre était dégénéré par la construction simple du TP précédent.

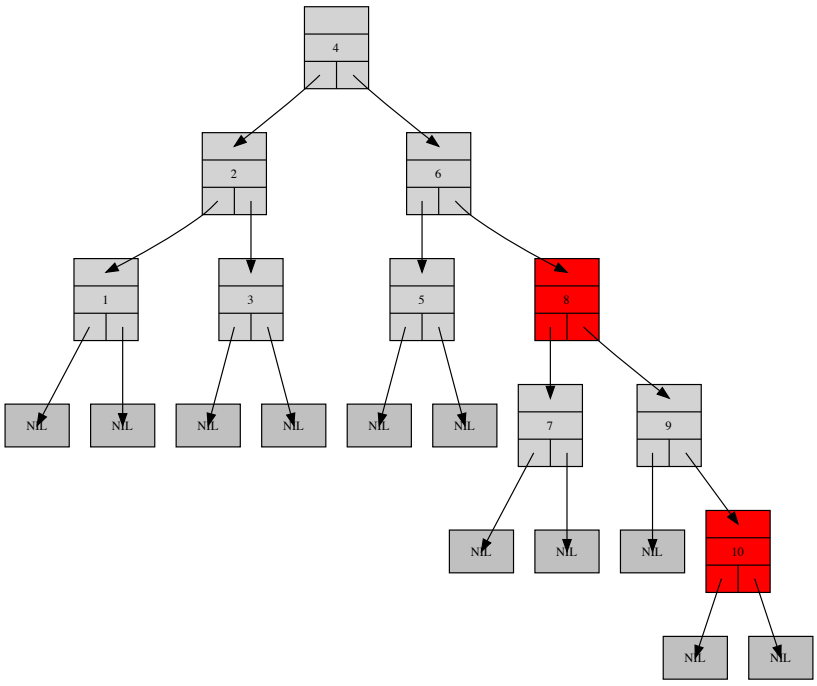

FIGURE 1 – Visualisation d'un arbre binaire de recherche équilibré

Le planning à suivre sur ces deux séances est le suivant :

Séance 1 Réalisation des exercices 1 à 4 inclus.

Séance 2 Réalisation de l'exercice 5.

1 Description de l'archive logicielle fournie

L'archive logicielle fournie pour ce TP comprend :

— Un module de gestion de file, `queue.h` et `queue.c`, implantant le TAD QUEUE vu en cours et déjà utilisé dans les TPs précédent.

— L'interface du module de gestion d'arbre binaire de recherche, `bstree.h`, utilisé au TP précédent et dont l'implantation a été développée dans le fichier `bstree.c`. Vous devez recopier votre implantation et l'étendre pour intégrer une gestion d'arbre rouge-noir. L'interface proposée est exactement la même que pour le TP précédent mais il a été ajouté les fonctions suivantes, permettant de réaliser les tests de votre implantation d'arbre Rouge-Noir.

```
/*----------------------  RBTSpecific  ----------------------*/
void rbtree_export_dot(const BinarySearchTree *t, FILE *file);
void testrotateleft(BinarySearchTree *t);
void testrotateright(BinarySearchTree *t);
```

1. https://www.graphviz.org/

— Un programme principal, `main.c`, effectuant les tests des différents exercices. Ce programme ne devra pas être modifié, seule la définition de constantes correspondant aux exercices effectués devront être décommentées.
— Un fichier `Makefile` permettant de compiler et de générer les fichiers pdf de visualisation des arbres.
— La documentation de l'archive fournie, à générer par la commande `make doc`.

Les développements à effectuer pendant ce TP ne concernent donc que le fichier `bstree.c` et l'implantation des fonctions indiquées dans le sujet. **Attention**, le résultat de ce TP servira de base de développement pour les TPs suivants. Une attention particulière devra donc être apportée sur la réutilisabilité et l'extensibilité du module développé.

Le but de ce TP est d'implanter les algorithmes d'équilibrage selon la méthode des arbres Rouge-Noir vue en cours.
Tous les développements demandés doivent être réalisés dans le fichier `bstree.c`. Pendant ce TP, aucune modification d'interface n'est nécessaire.

2 Propriétés des arbres Rouge-Noir

Les arbres Rouge-Noir (Red-Black Trees ou RBT) [2] sont une extension des arbres binaires de recherche permettant un équilibrage automatique de l'arbre après toute modification (ajout d'un élément, suppression d'un élément). Les arbres rouge-noir ne sont que "relativement équilibrés" mais assurent une complexité en $O(log(N))$ pour toutes les opérations de dictionnaire.
Les arbres rouge-noirs sont très adaptés à la manipulation de grandes collections de données dynamiques (nécessitant des ajouts/recherches/suppressions fréquentes) et sont à la base des implantations des structures associatives de nombreuses bibliothèques standard pour les différents langages de programmation (C++, Java, Python, ...).
Attention, la définition des arbres Rouge-Noir suppose que les informations (les clés) sont portées par les nœuds internes. Les feuilles, dénommées `NIL` dans la suite ne portent pas d'information.
Un arbre Rouge-Noir est un arbre binaire de recherche (possédant donc l'interface publique des BST vue précédemment) étendu dans lequel chaque nœud porte soit la couleur rouge, soit la couleur noire. Outre cette information de couleur ajoutée aux nœuds, un arbre Rouge-Noir vérifie les 3 propriétés suivantes :

1. Les feuilles (NIL), sont noires
2. Les fils d'un nœud rouge sont noirs
3. Le nombre de nœuds noirs le long d'une branche issue de la racine est indépendant de la branche. toutes les branches contiennent donc le même nombre de nœuds noirs.

Propriétés des arbres Rouge-Noir.

La première propriété est liée à la gestion particulière des feuilles. Les feuilles sont toutes égales à la constante (ou sentinelle) `NIL`, de couleur noire et ce choix de couleur permet d'assurer la vérification des autres propriétés pour tout arbre.
La seconde propriété permet de limiter le nombre de nœuds rouges présents dans un arbre.

2. Rudolf Bayer (1972). "Symmetric binary B-Trees : Data structure and maintenance algorithms". Acta Informatica. 1 (4) : 290–306. - https ://en.wikipedia.org/wiki/Red-black_tree

La troisième propriété, la plus importante, est une condition d'équilibrage de l'arbre. Elle indique que si l'on ne tient pas compte des nœuds rouges de l'arbre on obtient un arbre binaire de recherche parfaitement équilibré. Toutes les branches auront la même longueur.
Dans un arbre Rouge-Noir, on peut toujours considérer que la racine est noire. Si elle est rouge, on change sa couleur en noir et toutes les propriétés restent vérifiées. En effet, le fait de changer la couleur de la racine a un impact sur la longueur de toutes les branches. On peut donc considérer qu'une quatrième propriété existe : la racine de l'arbre est noire.

3 Coloration et rotations.

3.1 Exercice 1 : extension de la structure de données et coloration de l'arbre.

Afin de pouvoir gérer la couleur des nœuds de l'arbre, il faut rajouter à la structure de donnée définissant un nœud cette information de couleur. De plus, comme la structure du nœud n'est pas publique et pour pouvoir accéder à la couleur pour la visualisation de l'arbre, nous avons ajouté une fonction d'export de l'arbre coloré au format dot à l'interface `bstree.h`.

1. En ajoutant la définition de type `typedef enum {red, black} NodeColor;` à votre module, modifiez votre structure de données de façon à ajouter un champs `NodeColor color;` à tous les nœuds de l'arbre. Le type `NodeColor` doit-il être déclaré dans `bstree.h` ou dans `bstree.c` ?
2. Lors de la construction d'un nœud, initialiser cette couleur à la valeur `red`.
3. En prenant comme exemple la fonction `void node_to_dot(const BinarySearchTree *t, void *userData)` permettant de traduire un nœud au format dot dans le TP précédent, écrire la fonction `void printNode(const BinarySearchTree *n, void *out)` permettant de traduire un nœud de l'arbre Rouge-Noir en tenant compte de sa couleur. Pour faire apparaitre un nœud en rouge, il suffit de rajouter l'argument `fillcolor=red` dans la chaîne décrivant le format du nœud dot. Afin de pouvoir accéder à la représentation interne de vos nœuds, dans quel fichier devez vous écrire cette fonction ?
4. Ajouter dans votre fichier `bstree.c` la fonction suivante :

```
void rbtree_export_dot(const BinarySearchTree *t, FILE *file) {
    fprintf(file, "digraph RedBlackTree {\n\tgraph
       [ranksep=0.5];\n\tnode [shape = record];\n\n");
    bstree_iterative_depth_infix(t, printNode, file);
    fprintf(file, "\n}\n");
}
```

Après avoir programmé ces opérateurs, il faut alors décommenter la ligne `//#define EXERCICE_1` dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Vous devez obtenir l'affichage suivant :

```
$./redblack_trees ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Exporting the tree.

Done.
```

Après avoir transformé le fichier dot résultat de l'exécution (`redblacktree_0.dot` par la commande `make pdf`, vous devez obtenir l'arbre de la figure 2

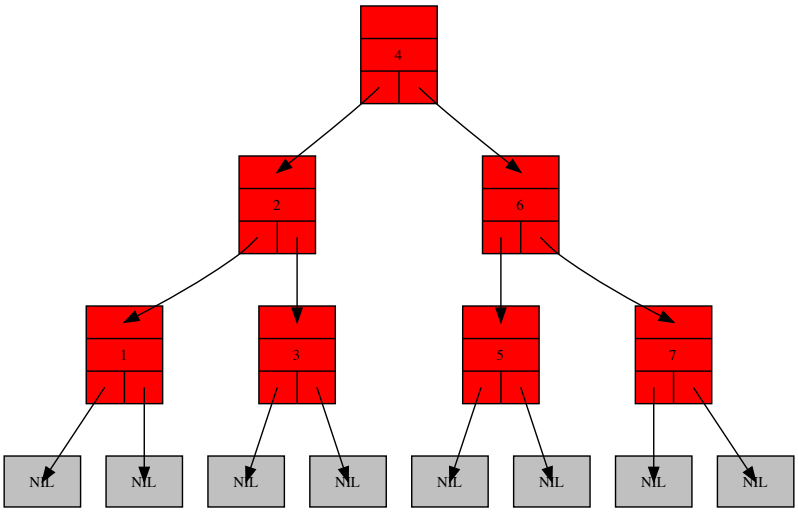

FIGURE 2 – Visualisation d'un arbre coloré en rouge.

3.2 Exercice 2 : programmation des opérateurs de rotation.

Dans un arbre binaire, une rotation est une opération locale sur un nœud permettant de diminuer la hauteur d'un de ses fils en augmentant celle de l'autre. Ces opérations locales ,et par conséquent réalisables en temps constant, sont à la base des opérations de rééquilibrage d'un arbre. Une rotation autour d'un nœud de l'arbre consiste à l'échanger avec un de ses fils (voir figure 3) en mettant à jour tous les liens nécessaires entre les nœuds.

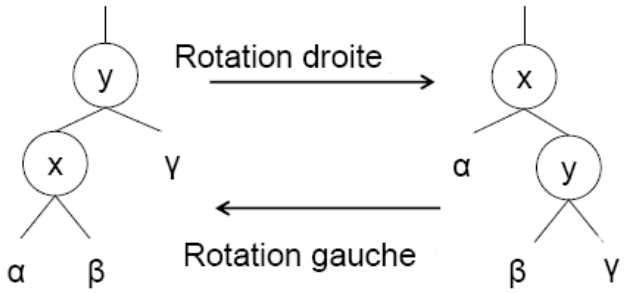

FIGURE 3 – Rotations dans un arbre.

Dans la rotation droite, un nœud devient le fils droit du nœud qui était son fils gauche. Par symétrie, dans la rotation gauche, un nœud devient le fils gauche du nœud qui était son fils droit. Les rotations gauche et droite sont l'inverse l'une de l'autre.

1. Ecrire les fonctions `void leftrotate(BinarySearchTree *x)` et `void rightrotate(BinarySearchTree *y)` effectuant une rotation gauche (respectivement droite) autour du nœud x (respectivement y). Ces fonctions doivent-elles être déclarées dans le fichier `bstree.h` ?
2. Ajouter dans votre fichier `bstree.c` les fonctions suivantes :

```
void testrotateleft(BinarySearchTree *t) {
    leftrotate(t);
}

void testrotateright(BinarySearchTree *t) {
    rightrotate(t);
}
```

Après avoir programmé ces opérateurs, il faut alors décommenter la ligne `//#define EXERCICE_2` dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Vous devez obtenir l'affichage suivant :

```
$./redblack_trees ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Exporting the tree.

Done.
Rotating the tree left around 4.
        Done.
Rotating the tree right around 4.

Done.
```

Après avoir transformé les fichiers dot résultat de l'exécution par la commande `make pdf`, vous devez obtenir les arbres de la figure 4

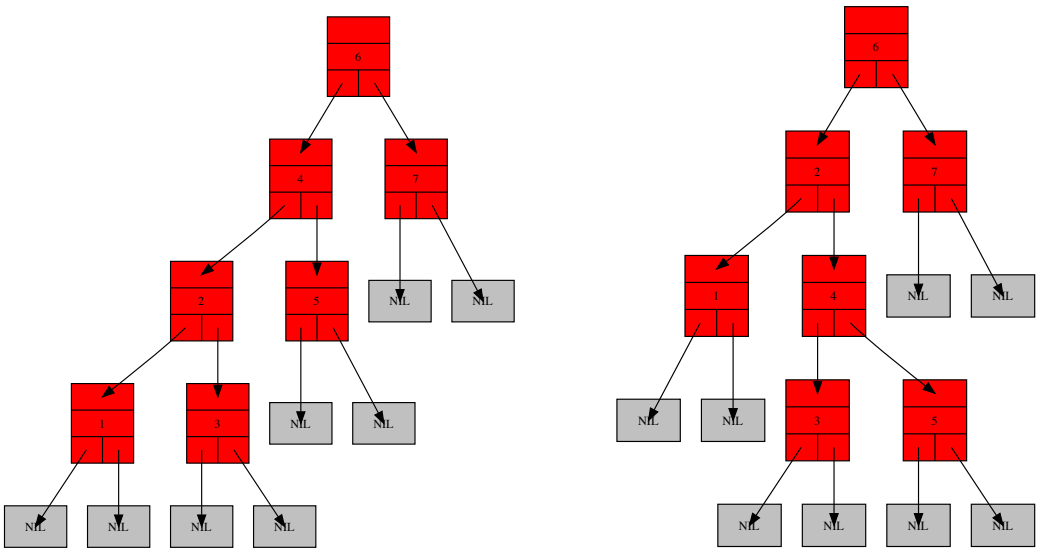

FIGURE 4 – Rotation gauche puis droite sur le nœud 4 à partir de l'arbre de la figure 1.

4 Insertion d'une valeur dans un arbre rouge-noir.

L'insertion d'une valeur dans un arbre Rouge-Noir est une des deux opérations pouvant éventuellement rompre les propriétés des arbres Rouge-Noir. L'implantation de cette opération se fait donc en deux étapes.

— Insertion de la nouvelle valeur
— Correction éventuelle des propriétés des arbres Rouge-Noir.

L'insertion de la nouvelle valeur dans l'arbre est faite de la même façon que l'insertion d'une valeur dans un arbre binaire de recherche. Cette insertion a lieu sur les feuilles. **Le nouveau nœud inséré est initialisé à la couleur Rouge**. Ce choix de coloration permet de conserver valide la propriété globale (3) des arbres Rouge-Noir et de simplifier l'éventuelle recoloration de l'arbre en permettant une implantation par analyse de cas locale. Il est à noter que le choix de conserver la propriété globale est important pour minimiser le coût de traitement.
Après l'insertion, si la propriété (3) est bien vérifiée, il n'en est pas nécessairement de même avec la propriété (2). Un simple test local, portant sur le père du nœud inséré permet de savoir si une recoloration est nécessaire. Si le père du nouveau nœud, inséré et coloré en rouge, est également rouge, la propriété (2) n'est plus vérifiée et une correction de la coloration de l'arbre doit être faite.
Le principe de construction de la fonction
`ptrBinarySearchTree fixredblackproperties_insert(ptrBinarySearchTree x)` de rétablissement des propriétés de l'arbre **au dessus** du nœud x repose sur une analyse exhaustive des configurations possibles de la coloration locale de l'arbre autour du nœud générant la violation de la propriété (2). Cette analyse nécessite d'accéder au nœud grand-parent et au nœud oncle d'un nœud de l'arbre.

4.1 Restauration des propriétés des arbres Rouge-Noir après insertion.

Soit x le nœud nouvellement coloré, et p son père qui sont tous les deux rouges. L'algorithme distingue plusieurs cas.

Cas 0 : le nœud père p de x est la racine de l'arbre.

Ce cas est illustré sur la figure 5. Il est à noter que ce cas peut être éliminé en coloriant systématiquement la racine de l'arbre en noir lorsqu'elle devient rouge. Intégrer un tel cas permet de ne faire cette coloration que lorsque c'est nécessaire.

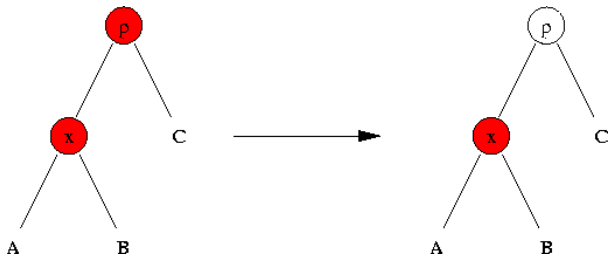

FIGURE 5 – Algorithme d'insertion : cas 0.

Si le nœud père p de x est la racine de l'arbre, il devient alors noir. La propriété (2) est maintenant vérifiée et la propriété (3) le reste aussi. L'algorithme d'insertion se termine. Ce cas est la seule condition générant une augmentation de la hauteur noire de l'arbre.
Si le nœud père p de x n'est pas la racine de l'arbre, alors, il faut traiter un des cas suivants.

Cas 1 : le frère f de p est rouge (f est donc l'oncle de x).

Ce cas est illustré sur la figure 6. On note pp le grand-père de x.

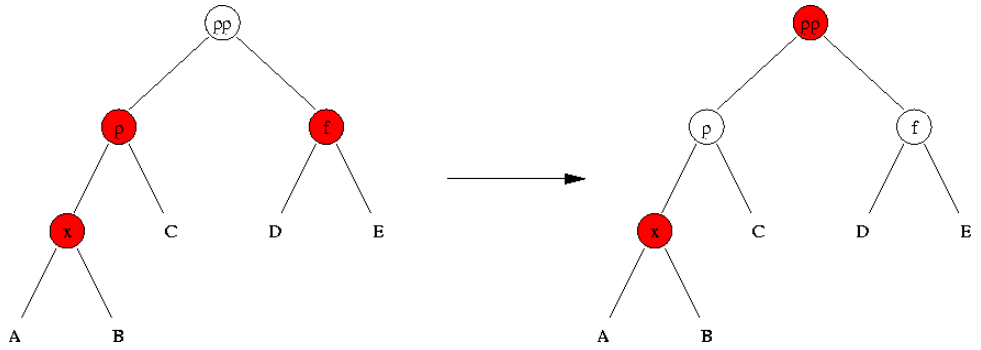

FIGURE 6 – Algorithme d'insertion : cas 1.

Si le frère f de p est rouge, les deux nœuds f et p deviennent noirs et leur père pp devient rouge. La coloration en noir de f et de p augmente de 1 la hauteur noire sur les deux branches correspondantes. La coloration en rouge du nœud pp, commun aux deux branches, diminue de 1 cette hauteur noire. La propriété (3) est donc toujours vérifiée. Si le père de pp est rouge, Il y a toujours 2 nœuds rouges qui se suivent dans l'arbre mais le problème a été décalé vers la racine. L'algorithme de recoloration de l'arbre continue alors en considérant pp comme le nouveau nœud coloré. Cette propagation vers la racine peut s'écrire simplement par un appel récursif (en récursivité terminale) à la fonction `fixredblackproperties_insert(pp)`. Il est à noter que c'est le seul cas de *boucle* de l'algorithme.

Cas 2 : le frère f de p est noir.

Afin d'étudier ce cas, nous considérons que le nœud p est le fils gauche de son père pp. Par symétrie, nous pourrons déduire l'algorithme à appliquer si p est le fils droit de pp.
Afin de traiter correctement ce cas, deux traitements différents doivent être réalisés selon que x est le fils gauche de p ou le fils droit de p.

1. Si x est le fils gauche de p, illustré sur la figure 7, l'algorithme effectue une rotation droite en pp. Le nœud p est coloré en noir et le nœud pp est coloré en rouge. L'algorithme se termine alors puisque les propriétés (2) et (3) sont maintenant vérifiées.

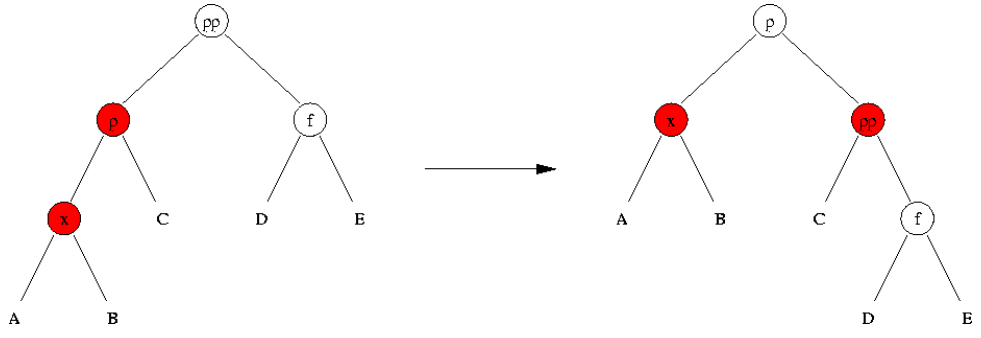

FIGURE 7 – Algorithme d'insertion : cas 2 - x est le fils gauche de p.

2. Si x est le fils droit de p, illustré sur la figure 8, l'algorithme effectue une rotation gauche en p de sorte que p devienne le fils gauche de x. On est alors ramené à la configuration précédente et on applique le même traitement.

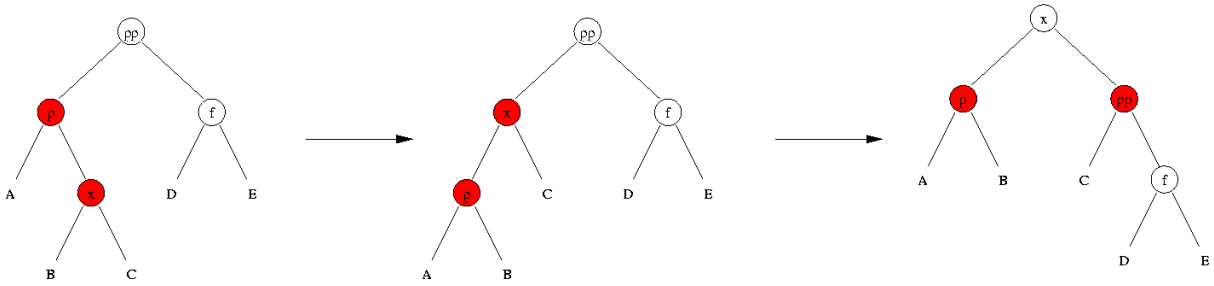

FIGURE 8 – Algorithme d'insertion : cas 2 - x est le fils droit de p.

4.2 Exercice 3 : programmation des opérateurs de correction après insertion.

En vous basant sur l'analyse de cas ci-dessus et expliquée en cours, et après avoir modifié votre fonction d'ajout d'une valeur dans l'arbre pour appeler l'opérateur de restauration des propriétés des arbres Rouge-Noir comme indiqué dans l'algorithme 1.

Listing 1 – Fonction d'insertion d'un nœud dans un arbre Rouge-Noir.

```
void bstree_add(ptrBinarySearchTree *t, int v) {
    /* Insert in the binary search tree t the value v and let cur be
       the new inserted node, colored in red */

    /* fix colors after insertion */
    BinarySearchTree *stop = fixredblack_insert(*cur);
    /* stop is the node at which the coloration procedure terminates.
       It may be the new root of the tree. */
    if (stop->parent == NULL)
        *t = stop;
}
```

Programmer les opérateurs suivants :

1. `ptrBinarySearchTree grandparent(ptrBinarySearchTree n);` qui retourne le grand-père du nœud n.
2. `ptrBinarySearchTree uncle(ptrBinarySearchTree n)` qui retourne l'oncle du nœud n.
3. `ptrBinarySearchTree fixredblack_insert(ptrBinarySearchTree x);` qui vérifie si le nœud x, de couleur rouge, nécessite une recoloration de l'arbre. Si aucune recoloration ne doit être faite, cet opérateur retourne le nœud x. Sinon, il retourne le résultat du traitement du cas 0 `return fixredblack_insert_case0(x)` à programmer ci-après.
4. `ptrBinarySearchTree fixredblack_insert_case0(ptrBinarySearchTree x)` qui traite le cas 0 de l'analyse de cas. Si une recoloration doit être faite, cette fonction retourne le résultat du traitement du cas 1 à programmer ci-après : `return fixredblack_insert_case1(x)`
5. `ptrBinarySearchTree fixredblack_insert_case1(ptrBinarySearchTree x)` qui teste si la configuration locale du nœud x correspond au cas 1 de l'analyse et applique le traitement correspondant. Après recoloration locale des nœuds, la propagation de la correction sur

le grand-père se fait par l'appel récursif terminal `return fixredblack_insert(pp);`. Si la configuration locale du nœud x ne correspond pas au cas 1, cet opérateur retourne le résultat de l'applicaiton du cas 2 développé ci-après : `return fixredblack_insert_case2(x);`

6. `ptrBinarySearchTree fixredblack_insert_case2(ptrBinarySearchTree x);` qui traite le cas 2 de l'analyse. Afin de prendre en compte la symétrie, vous pouvez écrire deux opérateurs symétriques `ptrBinarySearchTree fixredblack_insert_case2_left(ptrBinarySearchTree x);` et

   `ptrBinarySearchTree fixredblack_insert_case2_right(ptrBinarySearchTree x);` traitant les cas ou le père p de x est, respectivement, le fils gauche ou le fils droit de son père pp.

Après avoir programmé ces opérateurs, il faut alors décommenter la ligne `//#define EXERCICE_3` dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfile1.txt`.
Vous devez obtenir l'affichage suivant :

```
$./redblack_trees ../Test/testfile1.txt
Adding values to the tree.
        7 16 3 13 14 6 19 20 18 17 2 1 4 5 8 11 15 10 9 12
Done.
Exporting the tree.

Done.
```

Après avoir transformé les fichiers dot résultat de l'exécution par la commande `make pdf`, comparez l'arbre obtenu à celui que vous obteniez avec le même fichier de tests dans dans l'exercice 3 du TP précédent.

4.3 Exercice 4 : recherche de valeurs dans un arbre Rouge-Noir.

La recherche d'une valeur dans un arbre Rouge-Noir est indépendante de l'algorithme d'équilibrage et correspond exactement au même algorithme que pour la recherche dans un arbre binaire de recherche.
Afin de vérifier que vos opérations d'insertion se sont bien déroulées, décommenter la ligne `//#define EXERCICE_4` dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfile1.txt`.
Vous devez obtenir l'affichage suivant :

```
$./redblack_trees ../Test/testfile1.txt
Adding values to the tree.
        7 16 3 13 14 6 19 20 18 17 2 1 4 5 8 11 15 10 9 12
Done.
Exporting the tree.

Done.
Searching into the tree.
        Searching for value 5 in the tree : true
        Searching for value 14 in the tree : true
        Searching for value 11 in the tree : true
        Searching for value 23 in the tree : false
        Searching for value 71 in the tree : false
Done.
```

5 Suppression d'une valeur dans un arbre rouge-noir.

Comme pour l'insertion d'une valeur, la suppression d'un nœud dans un arbre rouge-noir commence par la suppression du nœud comme dans un arbre binaire de recherche.

- Si le nœud devant être supprimé possède 0 ou 1 fils, c'est ce nœud qui sera effectivement supprimé et son fils éventuel prend sa place.
- Si ce nœud possède 2 fils, il sera permuté avec son successeur dans l'arbre. Cette permutation nous ramène donc dans le cas précédent puisque le successeur ne possède au plus qu'un seul fils (le fils droit).

Remarquons d'abord que, si un nœud non vide (*i.e.* différent de NIL) est noir, alors son frère ne peut pas être une feuille.

Si le nœud supprimé est rouge, les propriétés des arbres Rouge-Noir restent vérifiées et aucune opération supplémentaire n'est nécessaire.

Si le nœud supprimé est noir, la propriété (3) est rompue et doit être restaurée.
Dans ce cas, le nœud remplaçant le nœud supprimé porte une couleur noire de plus pour rétablir la propriété (3). Ainsi, si le nœud remplaçant était rouge, il devient noir et l'algorithme se termine. Si le nœud remplaçant était noir, il devient alors "double-noir" et l'algorithme de suppression va appliquer une série de traitements afin de supprimer ce "double-noir", selon un principe similaire à la suppression du "double-rouge" effectuée lors de l'algorithme d'insertion. Par rotations successives, ce nœud "double-noir" va être remonté dans l'arbre jusqu'à absorption par un nœud rouge ou disparition par la racine.

Notons x le nœud "double-noir" devant être corrigé. Comme dans le cas de l'insertion, l'algorithme de correction de la coloration après suppression consiste à identifier les différentes configurations de l'arbre et à appliquer le traitement adapté. Toutefois, afin de préparer la recoloration éventuelle de l'arbre, cette fonction doit identifier le nœud à supprimer, son parent et le nœud de substitution. Attention, en raison de l'absence de sentinelle sur les feuilles dans notre implantation exemple, il est nécessaire de traiter de façon particulière la substitution par une feuille nulle. Ce traitement particulier nécessite de toujours connaître le père du nœud qu'on traite et de systématiquement tester la valeur du pointeur vers le nœud qu'on traite.

La fonction de suppression d'un nœud dans un arbre Rouge-Noir s'écrit alors comme une extension de la fonction de suppression d'un nœud dans un arbre binaire de recherche et correspond au listing 2.

Listing 2 – Fonction de suppression d'un nœud dans un arbre Rouge-Noir.

```
void bstree_remove_node(ptrBinarySearchTree *t, ptrBinarySearchTree
    current) {
        /* as for binary search trees, current is the node to delete
           and substitute replace the node to delete in the tree. */
    // search for substitute as when removing from a binary search
       tree.
    ptrBinarySearchTree substitute = ...;

    /* fix the redblack properties if needed */
    if (current->color == black) {
        if ( (substitute == NULL) || (substitute->color == black) ) {
            /* substitute is double black : must fix */
```

```
            ptrBinarySearchTree subtreeroot =
                fixredblack_remove(current->parent, substitute);
            if (subtreeroot->parent == NULL)
                *t = subtreeroot;
        } else {
            /* substitute becomes black */
            substitute->color = black;
        }
    }
    /* free the memory */
    free(current);
}
```

5.1 Restauration des propriétés des arbres Rouge-Noir après insertion.

Le principe de construction de la fonction
`ptrBinarySearchTree fixredblack_remove(ptrBinarySearchTree p, ptrBinarySearchTree x)` de rétablissement des propriétés de l'arbre **au dessus** du nœud x, dont le parent est p, repose sur une analyse exhaustive des configurations possibles de la coloration locale de l'arbre autour du nœud "double-noir" (le nœud x).

Cas 0 : le nœud x est la racine de l'arbre

Le nœud x devient noir. Il n'y a donc plus de nœud "double-noir" dans l'arbre et la propriété (3) reste vérifiée. Il est à noter que c'est le seul cas qui diminue la hauteur noire de l'arbre.
Si x n'est pas la racine de l'arbre, 2 cas seront considérés en fonction de la couleur du frère de x.

Cas 1 : le frère f de x est noir.

Afin d'étudier ce cas, nous considérons que le nœud x est le fils gauche de son père p et donc que f est le fils droit de p. Par symétrie, nous pourrons déduire l'algorithme à appliquer si x est le fils droit de p.
Le traitement de ce cas dépend de la couleur des fils du nœud f.

- Si les deux fils g et d de f sont noirs (figure 9), le nœud x devient noir, son frère f devient rouge et le père p porte une couleur noire de plus. Si p était rouge, il devient noir et l'algorithme s'arrête. Si p était noir, il devient alors "double-noir". L'arbre contient donc toujours un nœud "double-noir" mais celui-ci a été déplacé vers la racine. C'est le seul cas où l'algorithme de restauration continue.
- Si le fils droit d de f est rouge (figure 10), l'algorithme effectue une rotation gauche en p. Le nœud f prend la couleur de p, les nœuds x, p et d deviennent noirs et l'algorithme se termine.
- Si le fils droit d de f est noir et le fils gauche g de f est rouge (figure 11), l'algorithme effectue d'abord une rotation droite en f. le nœud g devient noir et le nœud f devient rouge. Comme la racine du sous-arbre D est nécessairement noire, il n'y a donc pas 2 nœuds rouges successifs dans l'arbre. On est donc ramené au cas précédent puisque maintenant le fils droit f du frère g de x est rouge. L'algorithme effectue donc une rotation gauche en p, f prend la couleur de p et et l'algorithme se termine comme précédemment.

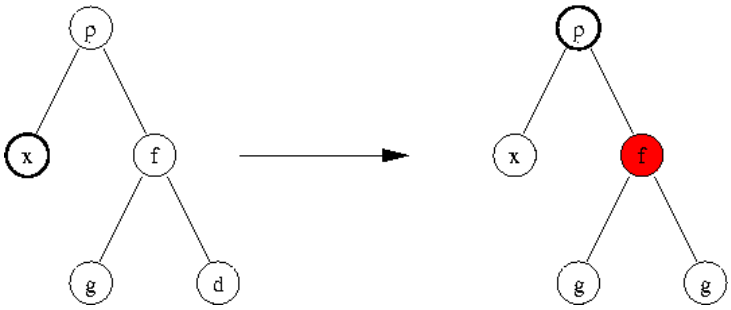

FIGURE 9 – Algorithme de suppression : cas 1 - les deux fils g et d de f sont noirs (le "double-noir" est indiqué par un cercle noir en gras sur le dessin).

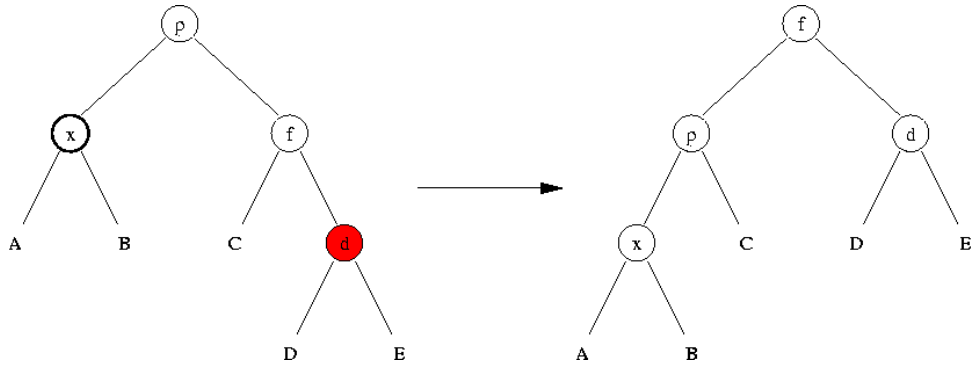

FIGURE 10 – Algorithme de suppression : cas 1 - le fils droit d de f est rouge.

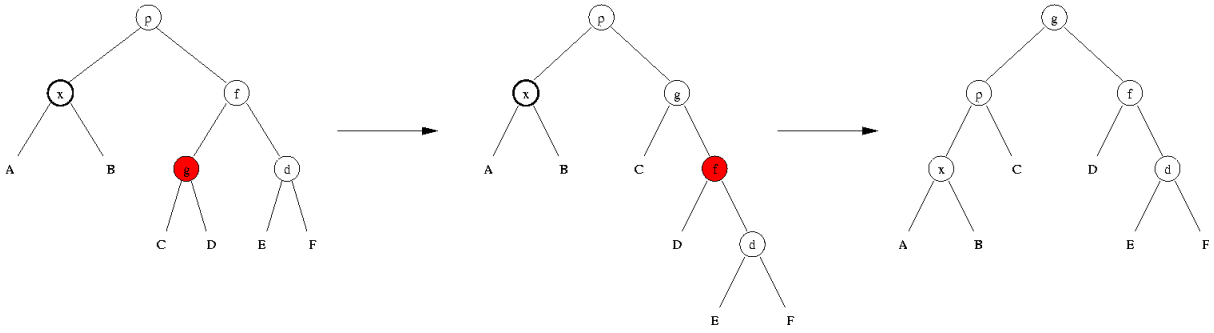

FIGURE 11 – Algorithme de suppression : cas 1 -le fils droit d de f est noir et le fils gauche g de f est rouge.

Cas 2 : le frère f de x est rouge.

Afin d'étudier ce cas, nous considérons que le nœud x est le fils gauche de son père p et donc que f est le fils droit de p. Par symétrie, nous pourrons déduire l'algorithme à appliquer si x est le fils droit de p.

Puisque f est rouge, le père p de f ainsi que ses deux fils g et d sont noirs. L'algorithme effectue alors une rotation gauche en p. Ensuite p devient rouge et f devient noir. Le nœud x reste doublement noir mais son frère est maintenant le nœud g qui est noir.

On est donc ramené au cas 1 ci-dessus.

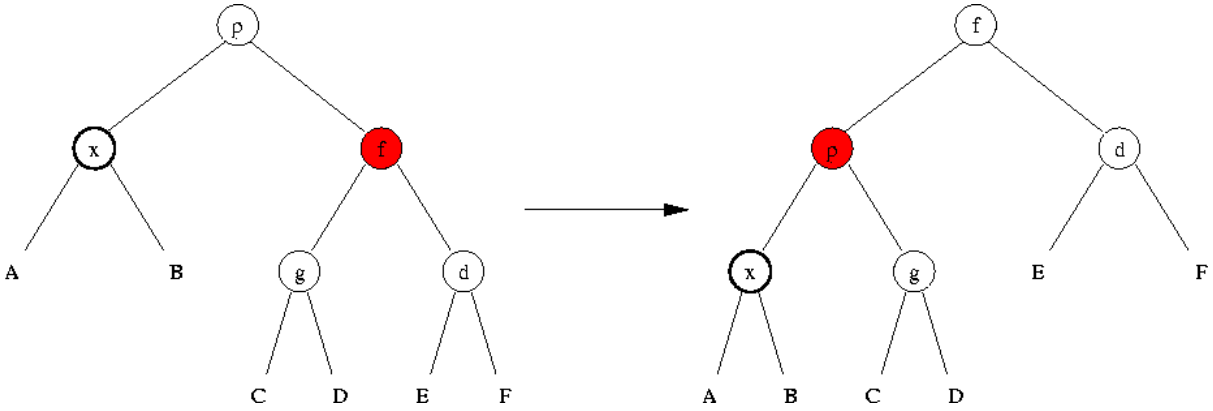

FIGURE 12 – Algorithme de suppression : cas 2 - x est le fils gauche de p et son frère f est rouge.

5.2 Exercice 5 : programmation des opérateurs de correction après suppression.

En vous basant sur l'analyse de cas ci-dessus et expliquée en cours, et après avoir modifié votre fonction de suppression d'une valeur dans l'arbre pour appeler l'opérateur de restauration des propriétés des arbres Rouge-Noir comme indiqué dans l'algorithme 2, programmer les opérateurs suivants :

1. `ptrBinarySearchTree fixredblack_remove(ptrBinarySearchTree p, ptrBinarySearchTree x)`. Cet opérateur traite le cas 0 ci-dessus et si ce cas n'est pas terminal, identifie le traitement a effectuer et retourne le résultats des opérateurs `fixredblack_remove_case1(p, x)` ou `fixredblack_remove_case2(p, x)`, programmés ci-dessous, et traitant respectivement la recoloration dans le cas 1 ou dans le cas 2 de l'analyse précédente.
2. `ptrBinarySearchTree fixredblack_remove_case1_left(ptrBinarySearchTree p, ptrBinarySearchTree x)`. Cet opérateur effectue une recoloration de l'arbre telle qu'expliquée dans le cas 1 de l'analyse en considérant que x est le fils gauche de son père p. Si le nœud "double-noir" est propagé vers la racine, i.e, c'est le parent p de x qui est maintenant "double-noir", cet opérateur renvoie le résultat `return fixredblack_remove(p->parent, p)`.
3. `ptrBinarySearchTree fixredblack_remove_case1_right(ptrBinarySearchTree p, ptrBinarySearchTree x)`. Cet opérateur traite le cas symétrique de l'opérateur précédent (x est le fils droit de son père p) et son code est alors identique à l'opérateur précédent, à la symétrie left/right près.
4. `ptrBinarySearchTree fixredblack_remove_case1(ptrBinarySearchTree p, ptrBinarySearchTree x)`. Cet opérateur retourne simplement l'appel à l'un des deux opérateurs précédents, selon que x est le fils gauche de p ou son fils droit.
5. Sur le même modèle que ci-dessus, mais pour traiter le cas 2 de l'analyse, programmez les opérateurs `ptrBinarySearchTree fixredblack_remove_case2_left(ptrBinarySearchTree p, ptrBinarySearchTree x)`, `ptrBinarySearchTree fixredblack_remove_case2_right(ptrBinarySearchTr p, ptrBinarySearchTree x)` et `ptrBinarySearchTree fixredblack_remove_case2(ptrBinarySearchTree p, ptrBinarySearchTree x)`.

Afin de vérifier que vos opérations d'insertion se sont bien déroulées, décommenter la ligne `//#define EXERCICE_4` dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Vous devez obtenir l'affichage suivant :

```
$./redblack_trees ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Exporting the tree.

Done.
Searching into the tree.
        Searching for value 2 in the tree : true
        Searching for value 5 in the tree : true
        Searching for value 8 in the tree : false
        Searching for value 1 in the tree : true
Done.
Removing from the tree.
        Removing the value 2 from the tree :
        Removing the value 5 from the tree :
        Removing the value 7 from the tree :
        Removing the value 3 from the tree :
        Removing the value 1 from the tree :
Done.
```

après avoir converti les fichiers dot produits en fichier pdf, vous devez obtenir les arbres suivants :

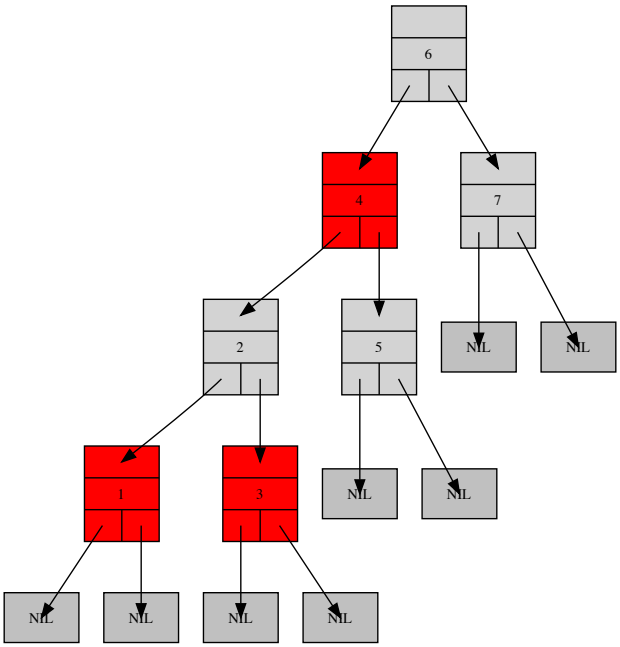

FIGURE 13 – Arbre construit avec le fichier testfilesimple.txt.

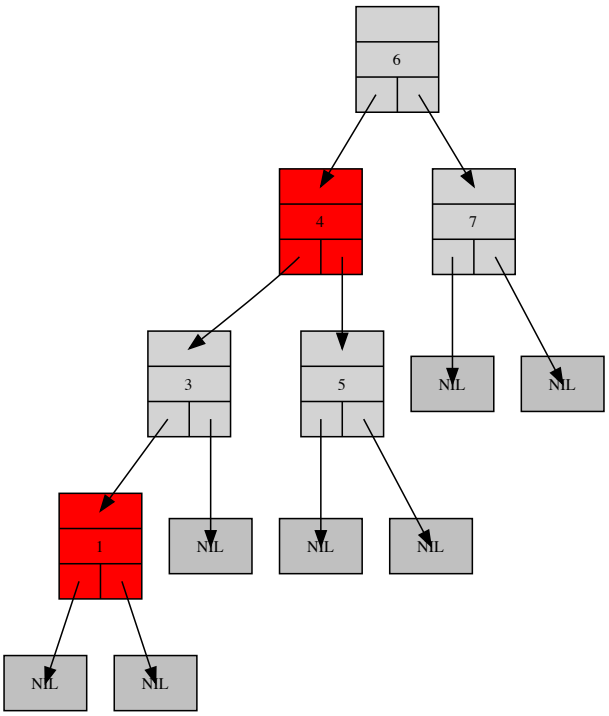

FIGURE 14 – Arbre après suppression de la valeur 2.

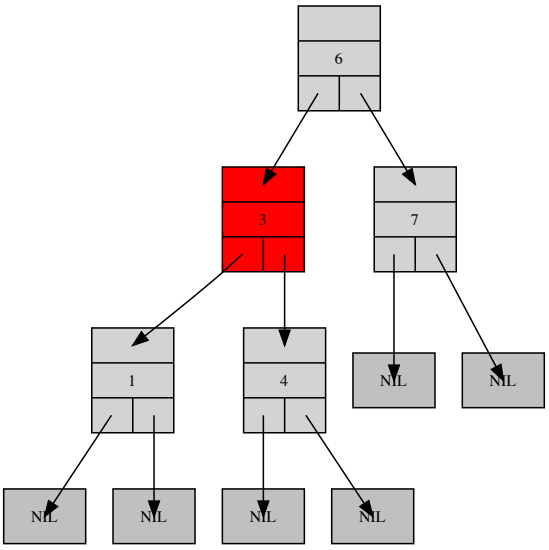

FIGURE 15 – Arbre après suppression de la valeur 5.

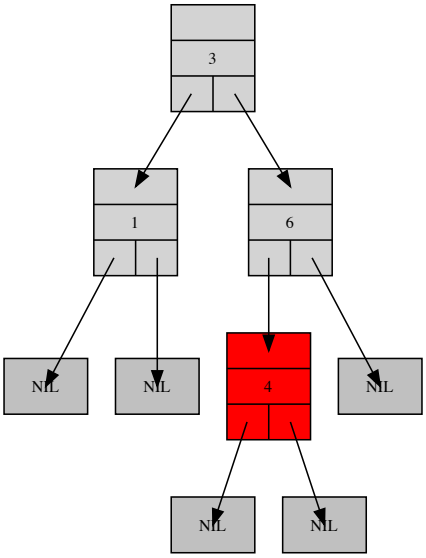

FIGURE 16 – Arbre après suppression de la valeur 7.

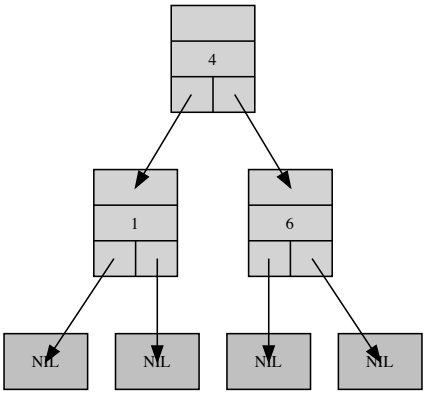

FIGURE 17 – Arbre après suppression de la valeur 3.

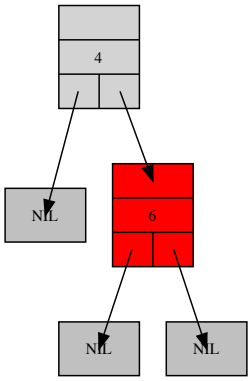

FIGURE 18 – Arbre après suppression de la valeur 1.