Arbres rouge-noir

L'objectif de ce sujet est d'obtenir une implémentation complète des arbres rouge-noir fonctionnels en OCaml : test d'appartenance, insertion et suppression. On utilisera le type suivant :

```
type 'a rn =
  | V
  | N of 'a rn * 'a * 'a rn
  | R of 'a rn * 'a * 'a rn
```

Les contraintes :
— Les étiquettes lues dans l'ordre infixe sont strictement croissantes.
— Un nœud rouge ne peut pas avoir de fils rouge.
— Tous les chemins de la racine à un nœud vide contiennent le même nombre de nœuds noirs.
— La racine est noire.
On appellera :
— *arbre rouge-noir correct* un arbre vérifiant les quatre conditions ci-dessus.
— *sous-arbre rouge-noir correct* un arbre vérifiant les trois premières conditions.
— *sous-arbre rouge-noir presque correct* un arbre vérifiant les trois premières conditions, sauf que sa racine peut être rouge et posséder un ou deux fils rouges.

1 Insertion

Pour insérer une nouvelle valeur, on l'insère de manière classique comme un arbre binaire de recherche et on colorie ce noeud en rouge. On corrige ensuite les problèmes en remontant jusqu'à la racine. On ne violera jamais la condition d'équilibre noir. Le seul problème potentiel sera un noeud rouge n avec un fils rouge et la résolution du problème sera la responsabilité du père (nécessairement noir) de n.

1. Dessiner les quatre cas problématiques possibles pour le père de n, et montrer que tous ces cas peuvent être résolus exactement de la même manière, de façon à obtenir un sous-arbre rouge noir correct dans lequel les hauteurs noires n'ont pas été modifiées.
2. Écrire une fonction `corrige_rouge` qui prend en entrée un arbre presque correct et :
   — effectue la transformation de la question précédente si c'est nécessaire (racine noire, un fils rouge qui a un fils rouge).
   — renvoie l'arbre tel que sinon.
   Cette fonction renverra un sous-arbre rouge-noir correct.

   ```
   corrige_rouge : 'a rn -> 'a rn
   ```

3. Écrire une fonction `insere_aux` qui prend en entrée un sous-arbre rouge-noir correct et renvoie un sous-arbre rouge-noir presque correct dans lequel la clé fournie a été insérée.

   ```
   insere_aux : 'a rn -> 'a -> 'a rn
   ```

4. Écrire la fonction `insere`, qui prend en entrée un arbre rouge-noir correct et une clé, et renvoie un arbre rouge-noir correct dans lequel la clé a été insérée.

   ```
   insere : 'a rn -> 'a -> 'a rn
   ```

2 Suppression

Le principe général de la suppression est le suivant :
— On commence par rechercher l'élément à supprimer. S'il n'est pas présent, il n'y a rien à faire.
— S'il a au plus un fils non vide, on le supprime.
— Sinon, on le remplace par son successeur (le minimum de son fils droit) et on supprime ce successeur.
— Dans les deux cas, on risque d'avoir introduit une violation de la condition d'équilibre noire.

— On déplace ce problème vers le haut de l'arbre, ou on le règle suivant les cas.
— En s'occupant de ce problème, on risque de violer la condition rouge-rouge, mais il sera toujours possible de rétablir immédiatement cette propriété.

1. **Cas de base pour la suppression.**
   Il y a quatre cas de base où l'on peut directement supprimer un élément, suivant que le nœud à supprimer est rouge ou noir et que son fils gauche ou droit est vide. On a représenté ci-dessous les deux cas correspondant à un fils gauche vide, avec les conventions suivantes (valables pour toute la suite) :
   — Les nœuds rouges sont en rouge **et en gras** (donc visibles après impression…).
   — Les arêtes rouges (celles menant à un nœud rouge) également.
   — Les nœuds noirs sont grisés.
   — Les arêtes noires sont en trait plein d'épaisseur normale.
   — Les arêtes en pointillés sont de couleur inconnue.

FIGURE 1 – Cas de base pour la suppression.

   Indiquer dans les deux cas le résultat de la suppression, en précisant si la hauteur noire a été modifiée ou non, et si oui, comment.

2.1 Suppression du minimum

On souhaite écrire une fonction `supprime_min` ayant la spécification suivante :

Entrées : un sous-arbre rouge-noir correct t, non vide ;

Sorties : un sous-arbre rouge-noir correct t' et un booléen b.

Post-conditions :
— étiquettes(t') = étiquettes$(t) \setminus \{\min t\}$;
— en notant h la hauteur noire de t et h' celle de t', on a soit $h' = h$, soit $h' = h - 1$;
— b est vrai si $h' = h - 1$, faux sinon.

Cette fonction va avoir la structure suivante :

```
let rec supprime_min arbre =
  match arbre with
  | V -> failwith "vide"
  | R (V, x, d) -> ...
  | N (V, x, d) -> ...
  | R (g, x, d) | N (g, x, d) ->
    let g', a_diminue = supprime_min g in
    ...
```

1. Compléter les lignes 4 et 5 de la fonction.

Quand on récupère le couple `g', a_diminue` (qu'il faut lire « a diminué » !), on ne peut pas *a priori* renvoyer `cons arbre g' x d` puisque la hauteur noire de `g'` peut être inférieure (de une unité) à celle de `d`. Il faut donc écrire une fonction permettant de rétablir l'équilibre noir dans ce cas. Les différents cas sont présentés ci-dessous :

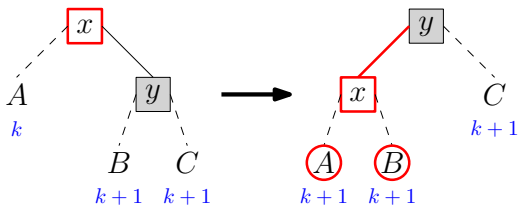

FIGURE 2 – repare noir gauche, racine rouge.

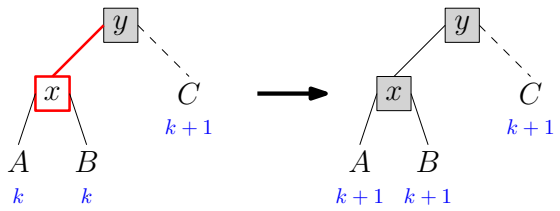

FIGURE 3 – repare noir gauche, racine noire et fils gauche rouge.

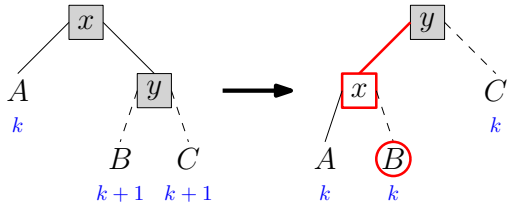

FIGURE 4 – repare noir gauche, racine noire, deux fils noirs.

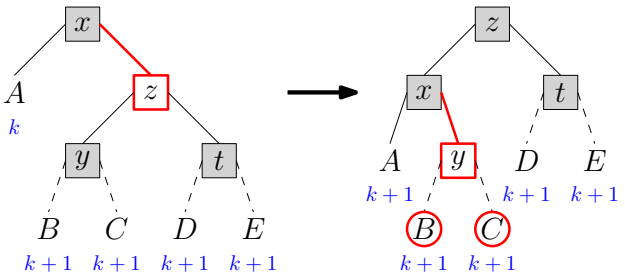

FIGURE 5 – repare noir gauche, racine noire et fils droit rouge.

2. Quelle est la signification des cercles rouges présents autour de certains nœuds dans les situations finales ?
3. Justifier que tous les cas sont présents, et bien traités.
4. Écrire la fonction `repare_noir_gauche`, dont la spécification est la suivante :

   Entrées : un arbre t et un booléen b.
   Sorties : un arbre t' et un booléen b'.
   Pré-conditions :
   — si b est faux, alors t est un sous-arbre rouge-noir correct ;
   — si b est vrai, alors t est de la forme (g, x, d) (la racine de t pouvant être rouge ou noire), g et d sont deux sous-arbres rouge-noir corrects, el la hauteur noire de d vaut exactement un de plus que celle de g.
   Post-conditions :
   — t' est un sous-arbre rouge-noir presque correct ;
   — les étiquettes de t' sont exactement celles de t ;
   — la hauteur de t' est
   — soit égale à celle de t, et dans ce cas b' vaut `false` ;
   — soit égale à celle de t moins un, et dans ce cas b' vaut `true`.
5. Compléter la fonction `supprime_min`.

```
supprime_min : 'a rn -> ('a rn * bool)
```

2.2 Suppression d'un élément quelconque

Quand on supprime un élément quelconque, on est amené à traiter le cas d'un arbre dont le fils *droit* possède une hauteur noire inférieure (de une unité) à celle du fils gauche : c'est par exemple le cas si la suppression du successeur a fait diminuer la hauteur du fils droit.

1. Représenter les cas symétriques de ceux des figures 2 à 5.
2. En déduire la fonction `repare_noir_droite`, « symétrique » de `repare_noir_gauche`.

```
repare_noir_droite : 'a rn -> bool -> ('a rn * bool)
```

3. Écrire une fonction `supprime_aux`, qui prend en entrée un sous-arbre rouge-noir correct t et une clé x et renvoie un couple (t', b) tel que :
   — t' est un sous-arbre rouge-noir correct ;
   — étiquettes(t') = étiquettes$(t) \setminus \{x\}$;
   — en notant h la hauteur noire de t et h' celle de t', on a
   — soit $h' = h$, et dans ce cas $b =$ `false` ;
   — soit $h' = h - 1$, et dans ce cas $b =$ `true`.

```
supprime_aux : 'a rn -> 'a -> ('a rn * bool)
```

4. Écrire finalement la fonction `supprime`, dont la spécification devrait aller de soi (elle doit renvoyer un arbre rouge-noir correct).

```
supprime : 'a rn -> 'a -> 'a rn
```