1 Introduction

L'objectif de ce sujet est d'obtenir une implémentation complète des arbres rouge-noir fonctionnels en OCaml : test d'appartenance, insertion, et suppression. Pour cela, on va travailler avec le type suivant :

```
type 'a rn =
  | V (* arbre vide *)
  | N of 'a rn * 'a * 'a rn (* noeud noir *)
  | R of 'a rn * 'a * 'a rn (* noeud rouge *)
```

Rappelons les contraintes que doit respecter un arbre rouge-noir :

- ce doit être un ABR (i.e. les étiquettes lues dans l'ordre infixe sont strictement croissantes) ;
- un nœud rouge ne peut pas avoir un 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 4 conditions ci-dessus ;
- **sous-arbre rouge-noir correct** un arbre vérifiant les 3 premières conditions (peut être pas la dernière) ;
- **sous-arbre rouge-noir *presque* correct** un arbre vérifiant les 3 premières conditions, sauf que sa racine peut être rouge et posséder un (ou deux) fils rouges.

1. Écrire une fonction `appartient : 'a rn -> 'a -> bool` prenant en arguments un arbre `t` et une valeur `x` et renvoyant `true` si `x` fait partie des étiquettes de `t` (et `false` sinon).

Il arrivera souvent que la couleur de la racine n'importe pas. Dans ce cas, plutôt que de faire deux fois la même chose, on pourrait utiliser une fonction construisant un arbre (g, x, d) dont la racine est de la même couleur que l'arbre de départ.

2. Écrire une fonction `cons : -> 'a rn -> 'a rn -> 'a -> 'a rn -> 'a rn` ayant la spécification suivante :

   Entrée : un arbre t non vide, un arbre g, une clé x, et un arbre d ;

   Sortie : un arbre t' tel que :
   - $t' =$ `N(g,x,d)` si la racine de t est noire ;
   - $t' =$ `R(g,x,d)` si la racine de t est rouge.

2 Insertion

Le principe est le suivant : on crée une nouvelle feuille rouge avec la clé à insérer, puis on corrige les problèmes en remontant jusqu'à la racine. Dans ce sujet, on ne fera pas explicitement de **rotations** mais on utilisera la puissance des **filtrages par motifs**.

Avec cette stratégie, on ne violera jamais la condition d'équilibre noir (troisième condition). Le seul problème potentiel sera un nœud rouge n avec un fils rouge, et la résolution de ce problème sera la responsabilité du père de n (qui est nécessairement noir).

Dans les schémas de ce sujet :

- les nœuds rouges et les arêtes menant à un nœud rouges sont en rouge et en gras (donc visible avec impression) ;
- les nœuds noirs sont grisés et les arêtes menant à un nœud noirs sont en trait plein et d'épaisseur normale ;
- les arêtes en pointillés mènent à des nœuds de couleur inconnue.

La figure 1 résume les 4 cas possibles pour le père de n, et comment les résoudre (toujours de la même manière) de façon à obtenir un sous-arbre rouge-noir **correct** dans lequel les profondeurs noires n'ont pas été modifiées.

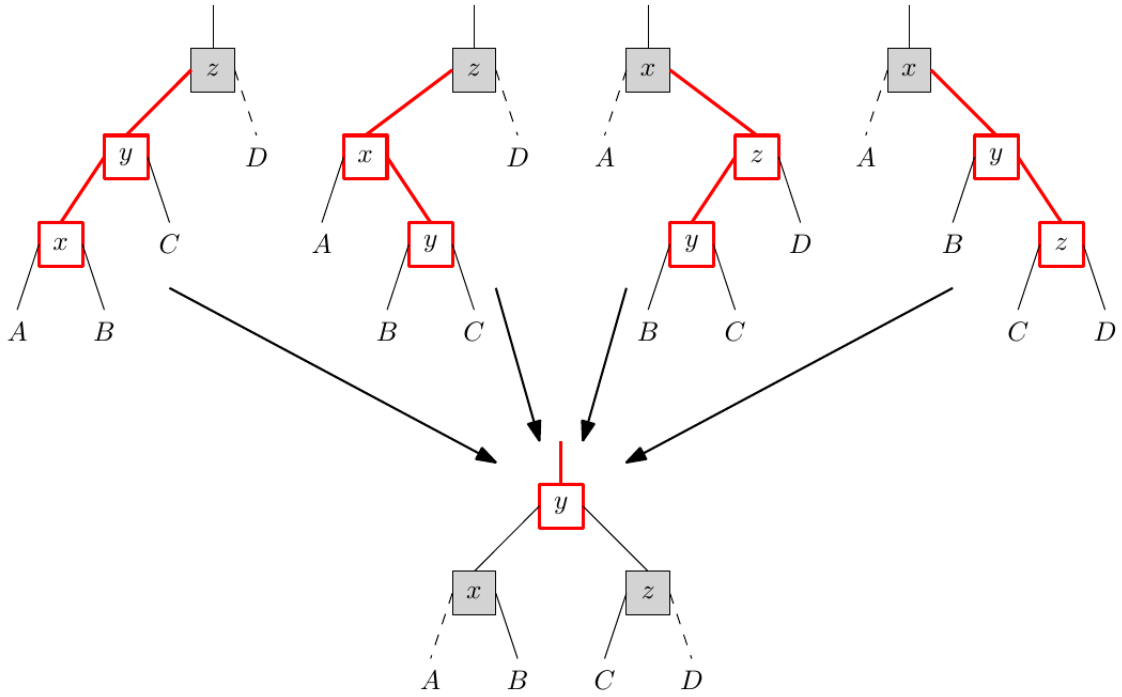

FIGURE 1 – Traitement des 4 cas rouge-rouge

3. Écrire une fonction `corrige_rouge : 'a rn -> 'a rn` qui prend en entrée un arbre et :
   - effectue la transformation de la figure 1 si c'est nécessaire (racine noire, et un fils rouge qui a un fils rouge) ;
   - renvoie l'arbre tel quel sinon.

   Cette fonction renverra un sous-arbre rouge-noir **presque correct**.
4. Écrire une fonction `insere_aux : 'a rn -> 'a -> 'a rn` qui prend en entrée un sous-arbre rouge-noir **correct** et une clé, et renvoie un sous-arbre rouge-noir **presque correct** dans lequel la clé fournie a été insérée.
5. Écrire une fonction `insere : 'a rn -> 'a -> 'a rn` qui prend en entrée un sous-arbre rouge-noir **correct** et une clé, et renvoie un sous-arbre rouge-noir **correct** dans lequel la clé fournie a été insérée.

3 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 (en renvoyant son éventuel fils non vide) ;
- 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 noir ;
- on déplace ce problème vers le haut de l'arbre, où on le règle suivant le 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.

3.1 Cas de base pour la suppression

Il y a 4 cas de base où l'on peut directement supprimer un élément x, suivant que le nœud à supprimer est rouge ou noir et que son fils gauche ou droit est vide. On a représenté en figure 2 les deux cas correspondant à un fils gauche vide.

FIGURE 2 – Cas de base pour la suppression

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

3.2 Suppression du minimum

On veut écrire une fonction `supprime_min : 'a rn -> ('a rn * bool)` ayant la spécification suivante :

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

Sortie : un sous-arbre rouge noir correct t' et un booléen b tels que :

- etiquettes(t') = etiquettes$(t) \setminus \{\min t\}$;
- 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`.

Cette fonction va avoir la structure suivante :

```
1 let rec supprime_min arbre = match arbre with
2     | V -> failwith "vide"
3     | R (V, x, d) -> ...
4     | N (V, x, d) -> ...
5     | R (g, x, d) | N (g, x, d) ->
6       let g', a_diminue = supprime_min g in
7       ...
```

7. Compléter les lignes 3 et 4 de la fonction `supprime_min`.

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 le profondeur 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 en figures 3 à 6.

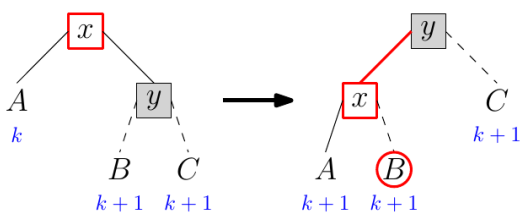

FIGURE 3 – `repare_noir_gauche`, racine rouge.

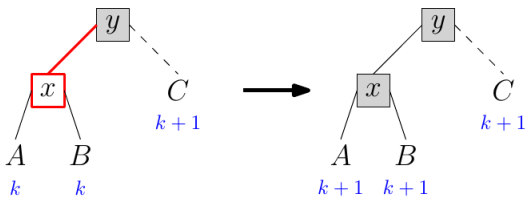

FIGURE 4 – `repare_noir_gauche`, racine noire et fils gauche rouge.

8. Quelle est la signification des cercles rouges présent autour de certains nœuds dans les situations finales ?
9. Justifier que tous les cas sont présents, et bien traités.

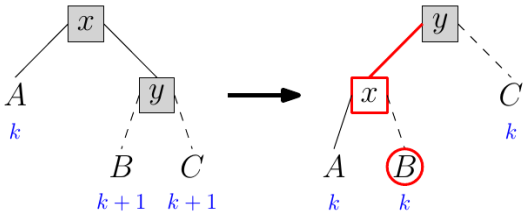

FIGURE 5 – `repare_noir_gauche`, racine noire, deux fils noirs.

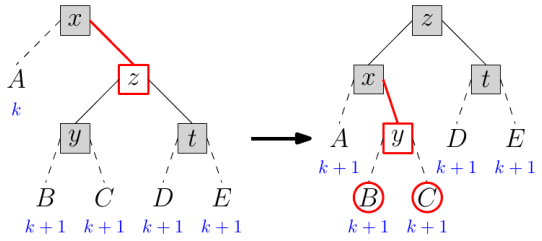

FIGURE 6 – `repare_noir_gauche`, racine noire et fils droit rouge.

10. Écrire une fonction `repare_noir_gauche : ('a rn * bool) -> ('a rn * bool)` dont la spécification est la suivante :

    Entrée : un arbre t et un booléen b tels que :
    - 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**, et la hauteur noire de d vaut exactement un de plus que celle de g ;

    Sortie : un arbre t' et un booléen b' tels que :
    - 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 1, et dans ce cas b' vaut `true` ;
11. Compléter la fonction `supprime_min`.

3.3 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.

12. Représenter les cas symétriques de ceux des figures 3 à 6.
13. En déduire la fonction `repare_noir_droite : 'a rn -> bool -> ('a rn * bool)` "symétrique" de `repare_noir_gauche`.
14. Écrire une fonction `supprime_aux : 'a rn -> 'a -> ('a rn * bool)` 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** ;
    - $\text{etiquettes}(t') = \text{etiquettes}(t) \setminus \{x\}$;
    - 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`.
15. Écrire finalement la fonction `supprime : 'a rn -> 'a -> 'a rn` qui supprime une clé dans un arbre rouge-noir **correct**, et renvoie un arbre rouge-noir **correct**.