TD XXVIII

ARBRES ROUGE-NOIR EN OCAML

L'objectif de ce sujet est d'obtenir une implémentation complète des arbres rouge-noir fonctionnels en OCaml : test d'appartenance, insertion et suppression. Nous allons utiliser une version des arbres rouge-noir légèrement différente de celle vue en cours (et nous ne réfléchirons pas en termes d'arbre 2-3).

Le type :

```
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.

Remarque

Dans le cours, nous avons utilisé des arbres rouge-noir *gauches*, dans lesquels on interdisait les fils droits rouges. Ce n'est pas le cas ici.

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 (et peut-être la dernière) ;
- *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

Le principe est essentiellement le même que dans le cours : on crée une nouvelle feuille rouge avec la clé à insérer, puis on corrige les problèmes en remontant jusqu'à la racine. Cependant, les cas à considérer sont un peu différents, et l'on ne fera pas *explicitement* de rotation : on remplacera cela par un usage massif du filtrage par motif.

Comme dans la méthode vue en cours, on ne violera jamais la condition d'équilibre noir. Le seul problème potentiel sera un nœud rouge n avec un fils rouge, et la résolution de ce problème sera la responsabilité du père (nécessairement noir) de n.

▶ **Question 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.

▶ **Question 2** Écrire une fonction `corrige_rouge` qui prend en entrée un arbre 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 presque correct.

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

▸ **Question 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
```

▸ **Question 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.

▸ **Question 5 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 XXVIII.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 :

```
1 let rec supprime_min arbre =
2   match arbre with
3   | V -> failwith "vide"
4   | R (V, x, d) -> ...
5   | N (V, x, d) -> ...
6   | R (g, x, d) | N (g, x, d) ->
7     let g', a_diminue = supprime_min g in
8     ...
```

▸ **Question 6** 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 :

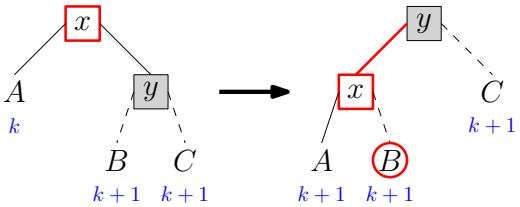


Figure XXVIII.2 – `repare_noir_gauche`, racine rouge.

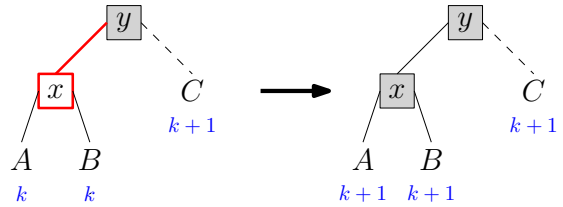


Figure XXVIII.3 – `repare_noir_gauche`, racine noire et fils gauche rouge.

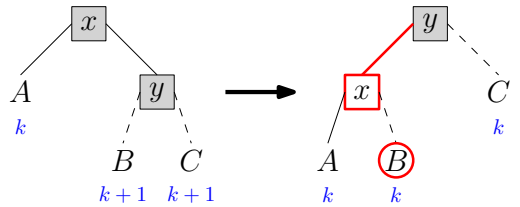


Figure XXVIII.4 – `repare_noir_gauche`, racine noire, deux fils noirs.

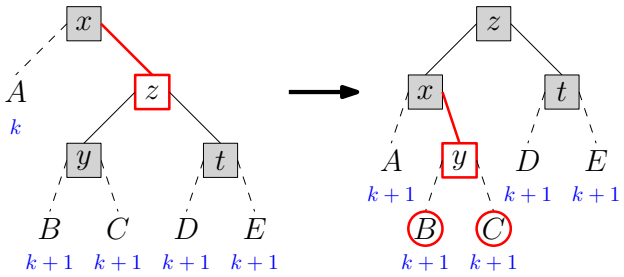


Figure XXVIII.5 – `repare_noir_gauche`, racine noire et fils droit rouge.

▸ **Question 7** Quelle est la signification des cercles rouges présents autour de certains nœuds dans les situations finales ?

▸ **Question 8** Justifier que tous les cas sont présents, et bien traités.

▸ **Question 9** 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`.

▶ **Question 10** 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.

▶ **Question 11** Représenter les cas symétriques de ceux des figures XXVIII.2 à XXVIII.5.

▶ **Question 12** En déduire la fonction `repare_noir_droite`, « symétrique » de `repare_noir_gauche`.

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

▶ **Question 13** 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 ;
- $\text{étiquettes}(t') = \text{étiquettes}(t) \setminus \{x\}$;
- 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)
```

▶ **Question 14** 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
```

Solutions

1 Insertion

▸ **Question 1**

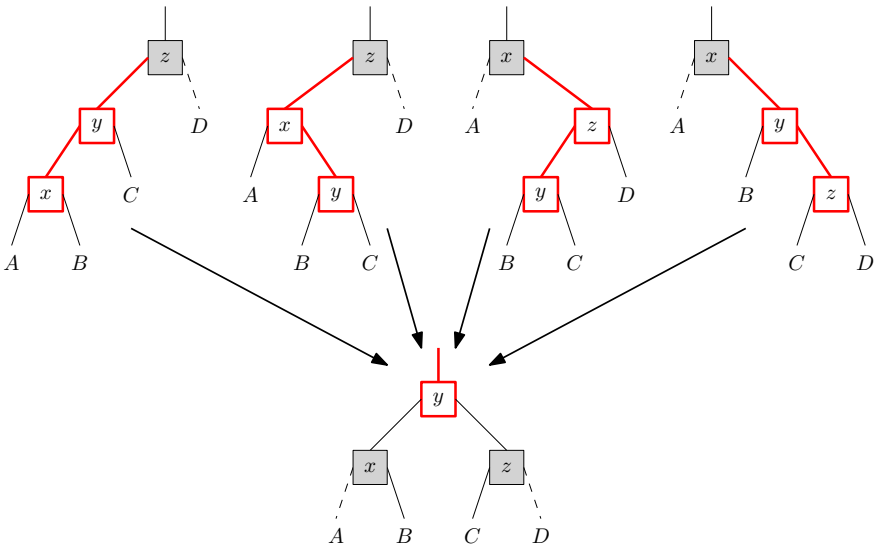


FIGURE XXVIII.6 – Traitement des quatre cas rouge-rouge.

▸ **Question 2** Sans le dessin, cette fonction peut faire un peu peur... Avec le dessin, en revanche, ça devrait être limpide.

```
let corrige_rouge = function
  | N (R (R (a, x, b), y, c), z, d)
  | N (R (a, x, R (b, y, c)), z, d)
  | N (a, x, R (R (b, y, c), z, d))
  | N (a, x, R (b, y, R (c, z, d)))
    -> R (N (a, x, b), y, N (c, z, d))
  | t -> t
```

▸ **Question 3** On utilise la fonction `cons` fournie pour éviter d'écrire deux fois la même chose. L'appel à `corrige_rouge` sera systématiquement inutile quand la racine de `t` est rouge, mais ce n'est pas grave (`corrige_rouge` ne fera simplement rien dans ce cas).

```
let rec insere_aux t x =
  match t with
  | V -> R (V, x, V)
  | R (l, y, r) | N (l, y, r) ->
    if x = y then t
    else if x < y then corrige_rouge (cons t (insere_aux l x) y r)
    else corrige_rouge (cons t l y (insere_aux r x))
```

▶ **Question 4** Il ne reste plus qu'à gérer la racine :

```
let noircit = function
  | R (g, x, d) -> N (g, x, d)
  | t -> t

let insere t x =
  noircit (insere_aux t x)
```

2 Suppression

▶ **Question 5** On renvoie A dans les deux cas. Dans le premier cas, la hauteur noire du nouvel arbre est un de moins que celle de l'arbre initial, dans le deuxième cas c'est la même.

2.1 Suppression du minimum

▶ **Question 6** Ces lignes correspondent aux cas de base de la question précédente :

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

Remarque
On avait parlé de cas symétriques à la question précédente : ils ne sont pas présents ici, puisque l'on supprime le *minimum* et qu'on a donc toujours un fils *gauche* vide.

▶ **Question 7** Les cercles rouges indiquent les nœuds susceptibles d'être rouges alors qu'ils ont un père rouge.

▶ **Question 8** On suppose qu'on part d'un sous-arbre rouge-noir correct, et que le sous-arbre gauche a une hauteur noire strictement plus petite que celle du sous-arbre droit. En particulier, cela signifie que la hauteur noire du fils droit est non nulle.

- Si la racine est rouge, les deux sous-arbres ont une racine noire, et le sous-arbre droit n'est pas vide : c'est notre premier cas.
- Si la racine est noire :
  - — soit le fils gauche est rouge (et donc non vide), ce qui correspond au deuxième cas ;
  - — soit les deux fils sont noirs, et le fils droit non vide d'après la remarque ci-dessus : c'est le troisième cas ;
  - — soit le fils gauche est noir et le fils gauche rouge. Les enfants du fils droit sont donc noirs, et comme la hauteur noire du fils droit est non nulle, ils ne peuvent pas être vides : c'est le quatrième cas.

Le fait que les cas soient correctement traités se vérifie facilement, surtout si l'on considère les annotations k / $k+1$ présentes sur les schémas. Les arbres obtenus en sortie vérifient la condition d'équilibre noir (dans le troisième cas, la hauteur noire a décru).

▶ **Question 9** À nouveau, c'est surtout une traduction du dessin. Il ne faut pas oublier d'appliquer `corrige_rouge` quand c'est nécessaire (et de l'appliquer au bon endroit, c'est-à-dire sur le père noir du nœùd rouge ayant un fils rouge). Remarquons que dans le dernier cas, on peut appeler `corrige_rouge` sur un nœud noir ayant un fils rouge dont *les deux fils* sont rouges. Nous n'avions pas considéré ce cas dans `corrige_rouge`, mais on vérifie facilement qu'il est correctement traité.

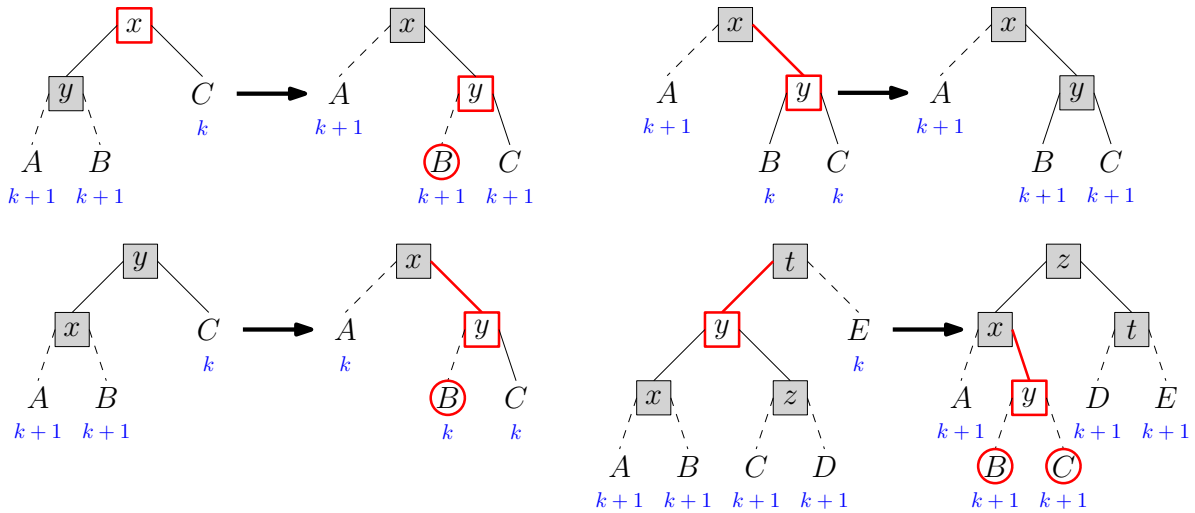
```
let corrige_noir_gauche arbre a_faire =
  if not a_faire then (arbre, false)
  else match arbre with
  | R (a, x, N (b, y, c)) ->
    corrige_rouge (N (R (a, x, b), y, c)),
    false
  | N (R (a, x, b), y, c) ->
    N (N (a, x, b), y, c),
    false
  | N (a, x, N (b, y, c)) ->
    corrige_rouge (N (R (a, x, b), y, c)),
    true
  | N (a, x, R (N (b, y, c), z, N (d, t, e))) ->
    N (corrige_rouge (N (a, x, R (b, y, c))),
       z,
       N (d, t, e)),
    false
  | _ -> failwith "impossible"
```

► **Question 10** Tout le travail a été fait : on pense juste à utiliser la fonction `cons` pour regrouper les deux cas identiques.

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

2.2 Suppression d'un élément quelconque

► **Question 11**



▶ **Question 12** On traduit les schémas :

```
let corrige_noir_droite arbre a_faire =
  if not a_faire then (arbre, false)
  else match arbre with
  | R (N (a, x, b), y, c) ->
    corrige_rouge (N (a, x, R (b, y, c))),
    false
  | N (a, x, R (b, y, c)) ->
    N (a, x, N (b, y, c)),
    false
  | N (N (a, x, b), y, c) ->
    corrige_rouge (N (a, x, R (b, y, c))),
    true
  | N (R (N (a, x, b), y, N (c, z, d)), t, e) ->
    N (corrige_rouge (N (a, x, R (b, y, c))),
       z,
       N (d, t, e)),
    false
  | _ -> failwith "impossible"
```

▶ **Question 13** Le plus gros du travail a déjà été fait, mais il faut quand même être soigneux.

```
let rec supprime_aux t x =
  match t with
  | V -> V, false
  | N (g, y, d) | R (g, y, d) when x < y ->
    let g', a_diminue = supprime_aux g x in
    corrige_noir_gauche (cons t  g' y d) a_diminue
 | N (g, y, d) | R (g, y, d) when x > y ->
    let d', a_diminue = supprime_aux d x in
    corrige_noir_droite (cons t g y d') a_diminue
  | N (V, _, t') | N (t', _, V) -> t', true
  | R (V, _, t') | R (t', _, V) -> t', false
  | N (g, _, d) | R (g, _, d) ->
    let m = minimum d in
    let d', a_diminue = supprime_min d in
    corrige_noir_droite (cons t g m d') a_diminue
```

▶ **Question 14** *Tout* le travail a déjà été fait !

```
let supprime t x =
  let t', _ = supprime_aux t x in
  noircit t'
```