TP 10 : Les arbres rouges-noirs

Le but de ce TP est de vous lancer dans l'implémentation des arbres rouge-noirs et comprendre par la pratique comment marche l'insertion à l'intérieur. Ce TP est sans doute l'un des plus compliqués de l'année : les arbres rouges-noirs demandent beaucoup de *soin*, de prendre le temps de temps de *tester* ses fonctions et de faire des *dessins*. Si ce n'est pas déjà le cas, il faut sortir un papier et un crayon (dont un rouge 😉). Dans la suite, on utilise les définitions suivantes :

Définition 1 – Définitions parallèles aux ARN

Soit A un arbre doté de couleurs.
- On dit que A est un ARN correct s'il vérifie toutes les conditions des ARN (c'est un ARN quoi).
- On dit que A est un sous-arbre rouge-noir correct s'il vérifie toutes les conditions des arbres rouge-noir, sauf la racine qui peut être rouge (mais ses fils sont noirs).
- On dit que A est un sous-arbre rouge-noir presque correct s'il vérifie toutes les conditions des arbres rouge-noir, sauf la racine qui peut être rouge (mais ses fils peuvent être rouges ou noirs).

Une autre définition évoquée à l'oral :

Définition 2 – Hauteur noire

La hauteur noire d'un arbre est le nombre de nœuds noirs dans un chemin de la racine à un nœud vide (sans compter le nœud vide). *Cette définition n'a de sens que pour les arbres rouges-noirs, pour lequel ce nombre est le même pour tous les chemins de la racine à un nœud vide.*

Exercice 1 – Introduction à l'implémentation que l'on choisis

Dans ce TP, on utilise le type suivant :

```OCaml
type couleur = Rouge | Noir
type 'a arn = V | N of couleur * 'a * 'a arn * 'a arn
```

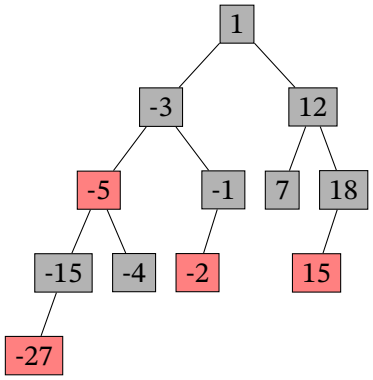


FIGURE 1 – Arbre rouge-noir correct

On le représente par le type `int arn` comme suis[a] :

```OCaml
let exemple = N ( Noir, 1,
                  N ( Noir, -3,
                      N ( Rouge, -5,
                        N (Noir, -15, N (Rouge, -27, V, V), V),
                        N (Noir, -4, V, V) ),
                      N (Noir, -1, N (Rouge, -2, V, V), V) ),
                  N ( Noir, 12,
                      N (Noir, 7, V, V),
                      N (Noir, 18, N (Rouge, 15, V, V), V) ) )
```

▶ **Question 1** Écrire une fonction `couleur_noeud : 'a arn -> couleur` renvoyant la couleur d'un nœud (`Rouge` si c'est un nœud rouge, `Noir` si c'est un nœud noir ou l'arbre vide). *Dans la suite, on choisira plutôt le filtrage par motifs pour reconnaître la couleur d'un arbre.*

▶ **Question 2** Reprendre le TP précédent pour écrire les fonctions suivantes :
- `min/max_arn : 'a arn -> 'a`
- `recherche : 'a arn -> 'a -> bool`

▶ **Question 3** En déduire une fonction vérifiant qu'un arbre représenté par le type `'a arn` respecte bien les conditions des ARN : on pourra commencer par écrire une fonction `est_ss_arn_correct` qui vérifie qu'un arbre est un sous-arbre rouge-noir presque correct.

a. Haha, maintenant je peux choisir quand je mets les numéros de ligne ou pas : vous pouvez copier et coller !

Exercice 2 – Insertion dans un arbre bicolore

On a commencé à en faire des dessins dans le cours, maintenant nous allons implémenter l'insertion de x dans un arbre rouge-noir. Contrairement au cours où l'on a décrit graphiquement l'insertion par l'utilisation de rotations, on va plutôt les implémenter *par des gros filtrages par motifs de leurs morts*. Pour rappel, notre stratégie est :
- On commence par parcourir la branche de l'arbre comme pour les arbres binaires de recherche classiques jusqu'à trouver le `V` dans lequel on pourrait l'insérer.
- On insère alors à la place de ce `V` un nœud **rouge** contenant x. L'arbre obtenu si l'on en restait là respecte la condition globale des ARN.
- Ce nœud rouge a peut-être un père n rouge. Si c'est le cas, le père de n est noir (puisque l'arbre dans lequel on a ajouté un élément est un arbre bicolore correct), et l'on doit réorganiser cet arbre de manière à faire remonter le nœud rouge au niveau du père de n (tout en conservant la condition globale des ARN).
- Si ce nœud rouge a pour père un nœud noir, on a fini ! En effet, le nombre de nœuds noir dans chaque branche a été conservé au cours de la remontée du nœud rouge et la condition locale est aussi respectée.
- Si ce nœud rouge se retrouve à la racine, il suffit de transformer ce nœud rouge en nœud noir et l'on a terminé (notez que seulement dans ce cas, la hauteur noire de l'arbre a augmenté).

Notez que la si on se retrouve avec un nœud n rouge avec un fils rouge, cela sera de la responsabilité du *père* de n de faire remonter le problème.

★ **Question 1** Rappeler sur papier ou au tableau quels sont les quatre configurations possibles d'un nœud n rouge dont l'un des fils est rouge et le père noir, et rappeler comment transformer cet arbre pour obtenir un arbre rouge-noir conservant la condition globale des ARN et faisant remonter l'éventuelle violation de la condition locale.

▶ **Question 2** Écrire une fonction `remonte_rouge : 'a arn -> 'a arn` effectuant cette opération : elle prend en entrée un arbre A et renvoie l'arbre transformé s'il se trouve dans l'une des quatre configurations de la Question 2.1, A inchangé sinon.

▶ **Question 3** En déduire une fonction `insere_aux : 'a arn -> 'a -> 'a arn` tel que pour l'appel `insere_aux a x`, pour `a` un sous-arbre rouge-noir correct et renvoie un sous-arbre rouge-noir presque correct contenant les mêmes étiquettes que `a` plus `x`.

▶ **Question 4** En déduire une fonction `insere : 'a arn -> 'a -> 'a arn` qui prend en entrée un ARN correct et renvoie un ARN correct dans lequel on a inséré une étiquette.

▶ **Question 5** Quelle est sa complexité ?

Dans la suite, on appelle *hauteur noire* la hauteur de l'arbre en ne comptant que les nœuds noirs.

Exercice 3 – Suppression

La suppression est une autre paire de manches : pour supprimer x d'un ARN A, on commence par le trouver similairement à la recherche. Quand on l'a trouvé, on a les cas faciles :

— S'il n'est pas présent (et donc que notre exploration nous amène à `V`), on renvoie l'arbre entier non modifié.
— Si l'un des deux fils est vide (ou les deux), on supprime le nœud et on renvoie l'autre fils (ou l'arbre vide).
— Si les deux fils sont non vides, on applique la seconde stratégie du dernier TP en supprimant le minimum du fils droit et en plaçant ce minimum en racine de l'arbre.

Mais Monsieur, mais Monsieur, oui, j'y viens, l'arbre obtenu dans ces deux derniers cas ne respecte pas forcément la condition globale sur les arbres rouges-noirs : ainsi, en remontant, il va falloir modifier les arbres rouges-noirs pour "rééquilibrer" ces arbres.

▶ **Question 1** On commence par considérer le cas où l'un des deux fils est vide (ou les deux), de la forme ci-dessous. Dans ces quatre cas, comparer la hauteur noire des deux arbres après suppression de la racine.

Passons à la suppression dans le cas général où les deux fils de x sont non vides. On doit d'abord être capable de supprimer le minimum du fils droit pour le mettre à la place de la racine, avec une fonction `supprime_min : 'a arn -> 'a arn * bool` qui prend en entrée un sous-arbre rouge-noir correct et qui renvoie un sous-arbre rouge-noir correct sans ce minimum tel que :
— Sa hauteur noire est la même que l'arbre en entrée, et b est faux,
— Sa hauteur noire est la hauteur noire de l'arbre en entrée moins un, et b est vrai.

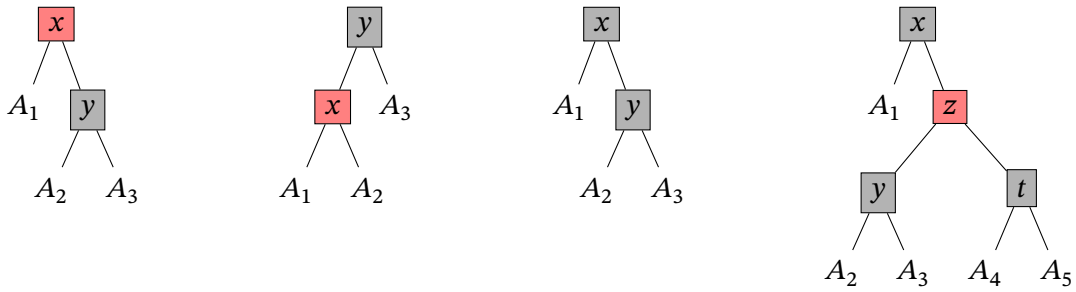
On complètera la fonction suivante :

OCaml
```
1 let rec supprime_min = function
2 | V -> V, false
3 | N (Rouge, e, V, d) -> ...
4 | N (Noir, e, V, d) -> ...
5 (* comme on supprime le minimum, les cas où le fils droit est vide ne nous
   ↪ intéressent pas *)
6 | N (c, e, g, d) ->
7 let new_g, b = supprime_min g in ...
```

▶ **Question 2** Compléter les lignes 2 à 4.

Dans le dernier cas, si `b` est faux, la hauteur noire de `new_g` est la même que `g`, donc on peut renvoyer l'arbre `N (c, e, new_g, d)` pour le motif ligne 6. Si au contraire `b` est vrai, la hauteur noire de `new_g` est égale à celle de `g` moins un, l'arbre n'est plus équilibré.

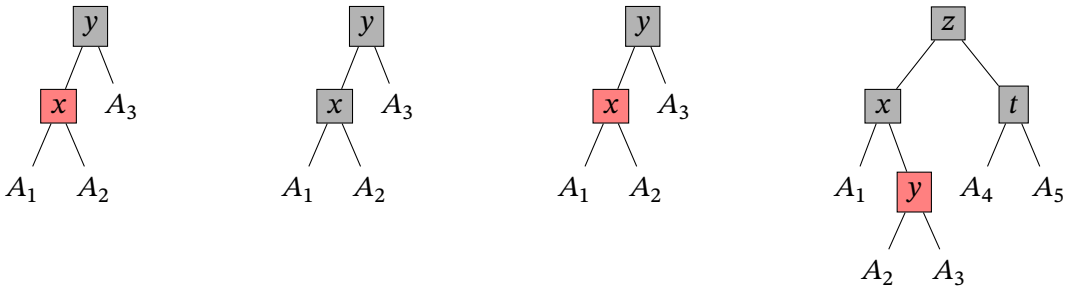
On considère les quatre cas suivants de configuration de l'arbre `N(c, e, new_g, d)` :



▶ **Question 3** Déterminer la hauteur noire de tous les arbres A_i dans les quatre cas, en fonction de celle de `new_g`.

▶ **Question 4** Justifier que seuls ces quatre cas sont possibles.

On propose les transformations suivantes pour équilibrer la hauteur noire de l'arbre `N (c, new_g, e, d)`.



▶ **Question 5** Dans certaines de ces transformations, la condition locale n'est potentiellement plus respectée. Déterminer quels cas et quels sous-arbres sont impactés. Quelle fonction déjà écrite permettrait d'y remédier ? Appliquée où ?

▶ **Question 6** Implémenter la fonction `corriger_gauche : 'a arn -> bool -> 'a arn * bool` répondant à la spécification suivante :

Entrée : Un arbre `a` et un booléen `b`

Précondition : si `b` est faux, `a` est un sous-arbre rouge-noir correct. Sinon, `a` est un arbre non vide dont les deux fils sont sous-arbre rouge-noir corrects et la hauteur noire de son fils gauche est exactement celle de son fils droit moins un.

Sortie : Un arbre `a'` et un booléen `b'`

Postcondition : `a'` est un sous-arbre rouge-noir presque correct dont les étiquettes sont les mêmes que `a`. De plus, `b'` est vrai si sa hauteur noire est la même que `a`, sinon `b'` est faux et sa hauteur noire est égale à celle de `a` moins un.

▶ **Question 7** En déduire une implémentation de la fonction `supprime_min : 'a arn -> 'a arn * bool`.

Pour supprimer un élément quelconque d'un ARN, on a dit qu'on utiliserait la seconde stratégie du dernier TP. Ainsi, on va être aussi amené à devoir rééquilibrer la hauteur noire d'un arbre dont la hauteur noire du fils droit a diminué.

▶ **Question 8** Déterminer les cas possibles pour la configuration de l'arbre `N (c, e, g, new_d)`, `new_d` étant le nouvel arbre obtenu après suppression d'une étiquette dans le fils droit.

▶ **Question 9** Reprendre les questions 3 à 6 jusqu'à obtenir une fonction `corriger_droite : 'a arn -> bool -> 'a arn * bool` répondant à la même spécification (sauf qu'on passe du fils gauche au fils droit étant potentiellement de hauteur noire plus petite en entrée).

▶ **Question 10** En déduire une fonction `supprime : 'a -> 'a arn -> 'a arn` implémentant la suppression. *On pourra utiliser une fonction auxiliaire de type* `'a arn -> 'a arn * bool` *qui prend en entrée un sous-arbre rouge-noir correct* `a` *et renvoie un couple* `a', b` *tel que* `a'` *contient les mêmes étiquettes que* `a` *sauf l'étiquette à supprimer et est de hauteur noire identique à* `a` *si* `b` *est faux, et celle de* `a` *moins un si* `b` *est vrai.*

Exercice 4 – Opérations non élémentaires sur les arbres rouges-noirs

On a vu que l'intérêt des arbres binaires de recherche en général est de permettre que certaines opérations parcourant théoriquement tous les éléments d'un ensemble aient une complexité plus petite que $\mathcal{O}(n)$: on peut citer évidemment la recherche, l'insertion et la suppression, la recherche du minimum et du maximum, la séparation d'un ensemble autour d'un pivot, etc.

Pour les ABR, ces opérations étaient en $\mathcal{O}(h(A))$. Si certaines opérations ne sont plus implémentables en $\mathcal{O}(h(A))$ en se limitant aux arbres rouges-noirs. Certaines autres sont surprenamment possibles :

▶ **Question 1** Écrire une fonction `joint : 'a arn -> 'a -> 'a arn -> 'a arn` tel que `joint a1 x a2`, pour `a1` un ARN ne contenant que des étiquettes strictement inférieures à `x` et `a2` un ARN ne contenant que des étiquettes strictement supérieures à `x`, renvoie un ARN contenant toutes les étiquettes de `a1`, `a2` et `x`. Sa complexité sera en $\mathcal{O}(|h(a_1) - h(a_1)|)$.

▶ **Question 2** En déduire une fonction `split : 'a arn -> 'a -> 'a arn` tel que `split a x` sépare l'ARN `a` en deux ARN, l'un contenant toutes les étiquettes de `a` inférieures à `x` et l'autre contenant toutes les étiquettes

supérieures à `x`. Sa complexité sera en $\mathcal{O}(h(A))$.

Ces deux fonctions permettent d'écrire d'autres opérations ensemblistes entre arbres rouge-noirs. Pour aller plus loin, on pourra regarder `http://www.cs.cmu.edu/~yihans/papers/join.pdf` par exemple.

▶ **Question 3** Reprendre le TP précédent en implémentant le *tri par arbre rouge-noir* triant une liste / un tableau : quelle est la complexité de votre algorithme dans le pire cas ?

En fait, l'algorithme de tri par ABR du dernier TP est en moyenne en $\mathcal{O}(n \log n)$, comme le permet de le voir le dernier exercice du dernier TP.