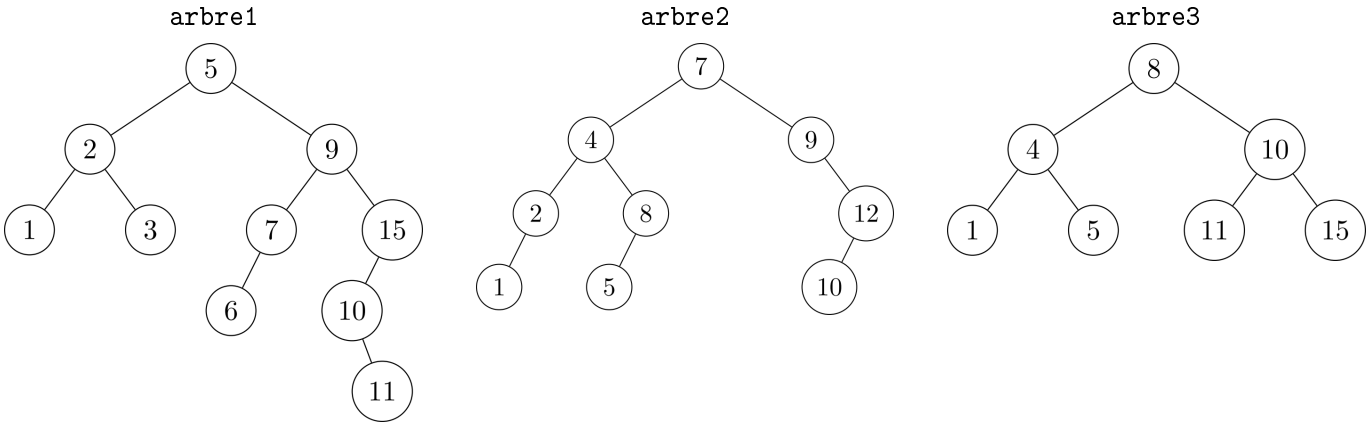
TP n°6 : Arbres binaires de recherche

1 Arbres binaires de recherche

Un *arbre binaire de recherche* est un arbre binaire dont les étiquettes appartiennent à un ensemble X ordonné et tel que pour tout nœud N, l'étiquette de N est supérieure ou égale à toutes les étiquettes présentes dans le fils gauche de N et inférieure ou égale à toutes les étiquettes présentes dans le fils droit de N.

Dans la suite, on utilisera des arbres binaires de recherche ne contenant que des étiquettes distinctes.

Question 1 Parmi les trois arbres suivants, lesquels sont des arbres binaires de recherche ?



Question 2 Définir un type `'a arbre` pour représenter un arbre binaire avec des étiquettes de type `'a`. Définir ensuite trois variables `arbre1`, `arbre2` et `arbre3` correspondant aux trois arbres précédents.

Question 3 Écrire une fonction `est_abr : 'a arbre -> bool` prenant en argument un arbre binaire et renvoyant `true` si cet arbre et un arbre binaire de recherche, `false` sinon.

Question 4 Écrire une fonction `contient : 'a -> 'a arbre -> bool` prenant en argument un élément x et un arbre binaire de recherche a et renvoyant `true` si l'étiquette x est présente dans a, `false` sinon.

Question 5 Quelle est la complexité de votre fonction `contient` ? Si celle-ci n'est pas dominée par la hauteur de l'arbre, la réécrire.

Question 6 Écrire une fonction `ajoute : 'a -> 'a arbre -> 'a arbr` prenant en argument un élément x et un arbre binaire de recherche a et renvoyant un arbre binaire de recherche dans lequel l'étiquette x a été ajoutée (si elle n'était pas déjà présente). Pour cela, on rajoutera une feuille bien placée dans l'arbre.

Question 7 La suppression d'un élément est plus délicate, car ce dernier peut correspondre à un nœud interne N. Dans ce cas :

- soit le fils gauche $\mathcal{A}_g$ de N est vide, et il suffit alors de remplacer N par son fils droit ;
- soit $\mathcal{A}_g$ n'est pas vide ; on cherche alors la plus grande étiquette de $\mathcal{A}_g$, qu'on supprime de $\mathcal{A}_g$ pour la mettre à la place de N.

Écrire une fonction `supprime_max : 'a arbre -> 'a * 'a arbre` prenant en argument un arbre binaire de recherche a et renvoyant un couple constitué de sa plus grande étiquette et de l'arbre binaire de recherche obtenu en supprimant celle-ci dans a.

Question 8 Écrire une fonction `supprime : 'a -> 'a arbre -> 'a arbre` prenant en argument un élément x et un arbre binaire de recherche a et renvoyant l'arbre binaire de recherche obtenu en supprimant x dans a.

Question 9 Écrire une fonction `tri_abr : 'a list -> 'a list` qui trie une liste ℓ en construisant un arbre binaire de recherche contenant tous les éléments de ℓ puis en utilisant un parcours d'arbre bien choisi.

2 Arbres rouges et noirs

La complexité des opérations sur un arbre binaire de recherche dépend de la hauteur de l'arbre. Il est donc préférable d'avoir des arbres binaires de recherche *équilibrés*, c'est-à-dire pour lesquels la hauteur est un $O(\log n)$ où n est le nombre de nœuds de l'arbre.

Un *arbre rouge et noir* est un arbre binaire de recherche dont chauqe nœud possède une couleur qui vaut soit rouge, soit noir, et qui vérifie les deux propriétés suivantes : 1. Les fils d'un nœud rouge sont noirs ; 2. Le chemin entre la racine et n'importe quelle feuille contient toujours le même nombre nœud noir.

On utilisera les types suivants :

```
type couleur =
  | Rouge
  | Noir
;;

type 'a arbreRN =
  | Vide
  | Noeud of couleur * 'a * 'a arbreRN * 'a arbreRN
;;
```

2.1 Vérification des propriétés

Question 10 Écrire une fonction `verifier_couleur : 'a arbreRN -> bool` prenant en argument un élément a de type `arbreRN` et renvoyant le booléen `true` si a vérifie la propriété 1 (i.e tout fils d'un nœud rouge est noir), `false` sinon.

Question 11 On appelle *hauteur noire* d'un arbre rouge et noir le nombre de nœuds noirs présents dans n'importe quel chemin de la racine à une feuille.

La hauteur noire d'un élément de type `arbreRN` n'étant défini que lorsque la propriété 2 est vérifiée, on va utiliser le type `'a option` présent dans la bibliothèque standard OCaml, qui peut être décrit de la manière suivante :

```
type 'a option =
  | Some of 'a
  | None
;;
```

Écrire une fonction `hauteur_noire : 'a arbreRN -> int option` prenant en argument un arbre de type `arbreRN` et renvoyant la hauteur noire de l'arbre si la propriété 2 est vérifiée, `None` sinon.

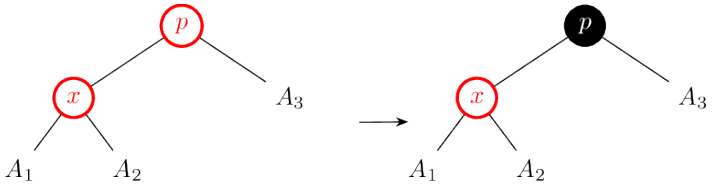
2.2 Insertion d'un élément

Pour insérer un élément dans un arbre rouge et noir, on l'insère en procédant comme dans un arbre binaire de recherche classique en donnant au nœud la couleur rouge (afin de préserver la propriété 2), puis on réarrange et on recolorie les nœuds afin de faire disparaître les éventuels conflits relatifs à la propriété 1.

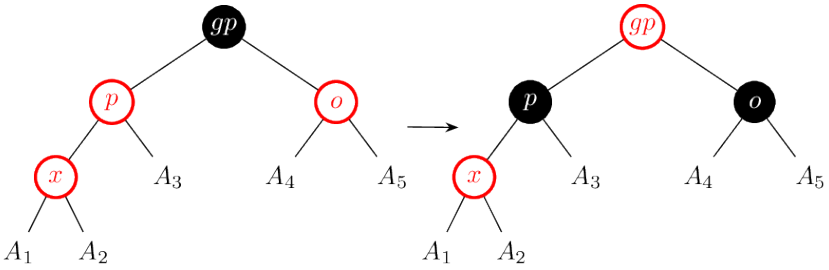
Question 13 Écrire une fonction `insertion_simple : 'a -> 'a arbreRN -> 'a arbreRN` qui prend en argument un élément x de type `'a` et un arbre rouge et noir a et qui renvoie l'arbre obtenu en insérant x dans l'arbre a en conservant la structure d'arbre binaire de recherche et en le coloriant en rouge (mais sans se préoccuper de la propriété 1).

Question 14 Lorsque la propriété 1 n'est plus vérifiée, c'est-à-dire lorsqu'un nœud x et son père p sont tous les deux rouges, on appliquera une des transformations suivantes, dont on vérifiera qu'elles conservent la structure d'arbre binaire de recherche ainsi que la propriété 2 :

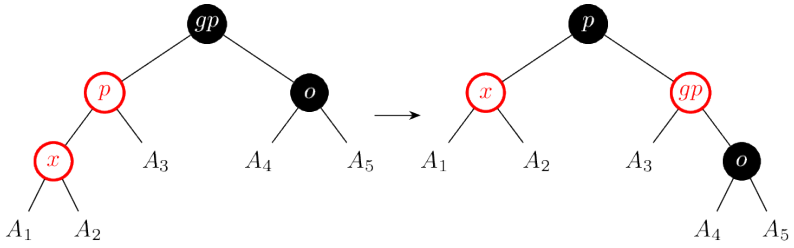
- Si p est la racine de l'arbre, on la colorie en noir.



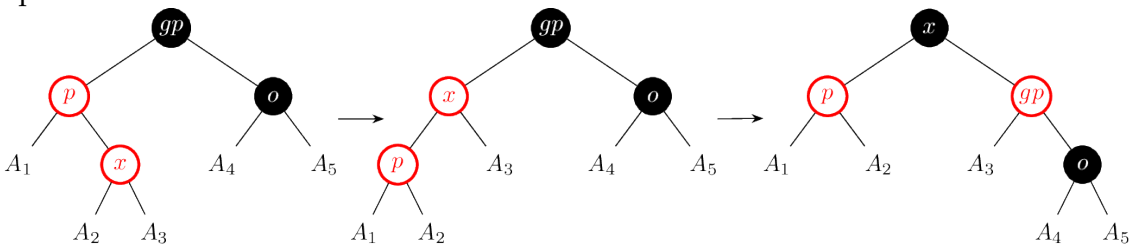
- Si le frère o de p est rouge, on colorie p et o en rouge et leur père gp en noir. La propriété 1 n'est pas forcément vérifiée, mais le conflit s'est déplacé vers la racine.



- Si p est le fils gauche de son père gp :
  - lorsque x est le fils gauche de p, on effectue une rotation droite entre p et gp, et on colorie p en noir et gp en rouge.



  - lorsque x est le fils droit de p, on effectue d'abord une rotation gauche entre x et p, ce qui ramène au cas précédent.



- Si p est le fils droit de son père : on procède de manière symétrique au cas précédent.

Écrire une fonction `insere_RN : 'a -> 'a arbreRN -> 'a arbre RN` qui prend en argument un élément x de type `'a` et un arbre rouge et noir a et qui renvoie l'arbre obtenu en insérant x dans l'arbre a en conservant la structure d'arbre rouge et noir.