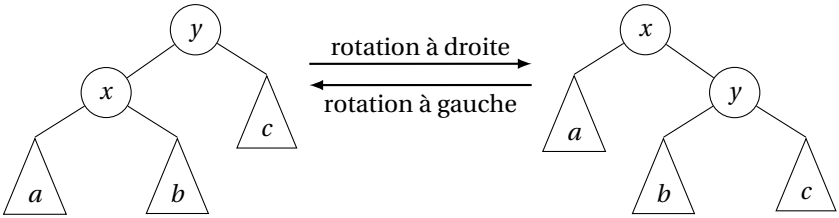
Arbres rouges-noirs

I. Introduction

On définit les opérations de rotation à gauche et à droite de la façon suivante :



Question 1 Montrer que l'application d'une rotation (à gauche ou à droite) à n'importe quel nœud d'un a.b.r. préserve la structure d'a.b.r.

Il s'agit de l'opération de base pour équilibrer les arbres binaires de recherche.

II. Arbres rouges-noirs

Nous allons définir, puis étudier, une structure d'arbre binaire qui, sont des conditions assez simples à définir et à maintenir, permet d'avoir des arbres binaires de recherche équilibrés.

Définition

☞ Un **arbre rouge-noir** est un arbre binaire de recherche où :

- ⋆ tous les nœuds internes sont colorés en rouge ou en noir,
- ⋆ toutes les feuilles sont colorées en noir.

Lorsque l'on parlera de la couleur d'un arbre, on fera référence à la couleur de sa racine.

☞ Un arbre rouge-noir est dit **équilibré** si la coloration de l'arbre vérifie les règles suivantes :

(R_1) la racine est noire ;

(R_2) les fils d'un nœud rouge sont nécessairement noirs ;

(R_3) tous les chemins allant de la racine à une feuille rencontrent le même nombre de nœuds noirs.

Pour la suite, on définit les types suivants :

```
type color = Red | Black

type 'a rbt = Leaf | Node of color * 'a rbt * 'a * 'a rbt
```

Nous allons commencer par montrer que la hauteur d'un arbre rouge-noir équilibré est en $O(\log(n))$ où n est le nombre de nœuds de l'arbre. Pour cela, soit un arbre rouge-noir ayant n nœuds. Pour tout nœud (interne ou feuille) v de l'arbre, on note :

☞ $h(v)$ la hauteur du sous-arbre enraciné en v, et

☞ $h_n(v)$ sa « *hauteur noire stricte* », autrement dit le nombre commun de nœuds noirs traversés entre v et ses feuilles-filles, sans compter v même s'il est noir.

Question 2 Montrer que pour tout nœud v de l'arbre, le sous-arbre enraciné en v a au moins $2^{h_n(v)} - 1$ nœuds internes.

Question 3 En déduire que la hauteur h de l'arbre vérifie :

$$h \leq 2\log_2(n+1) + 1$$

Ainsi, si un arbre rouge-noire à sa coloration qui vérifie les trois règles, on est assuré que sa hauteur est en $P(\log(n))$.

III. Implémentations, échauffement

Question 4 Écrire une fonction

```
mem : 'a -> 'a rbt -> bool
```

telle que `mem v t` renvoie `true` si la valeur `v` est présente dans `t`, et `false` sinon. On s'assurera que si `t` est équilibré, alors `mem` a une complexité en $O(\log|t|)$.

Question 5 Écrire une fonction

```
min : 'a rbt -> 'a
```

qui renvoie la valeur minimale contenue dans l'arbre, et lève une exception si le minimum n'est pas défini.

Dans les fonctions suivantes, sauf mention contraire,
on suppose que l'arbre passé en argument est un arbre rouge-noir équilibré.

IV. Insertion dans un arbre rouge-noir

Pour insérer une nouvelle valeur dans un arbre, on commence par effectuer une *insertion aux feuilles*, le nouveau nœud inséré étant de couleur rouge pour respecter la règle (R_3). Cependant, la règle (R_2) peut, elle, ne plus être vérifiée, et nous allons corriger cela en faisant « remonter » l'éventuel conflit jusqu'à la racine, pour le faire finalement disparaitre.

Question 6 Écrire une fonction

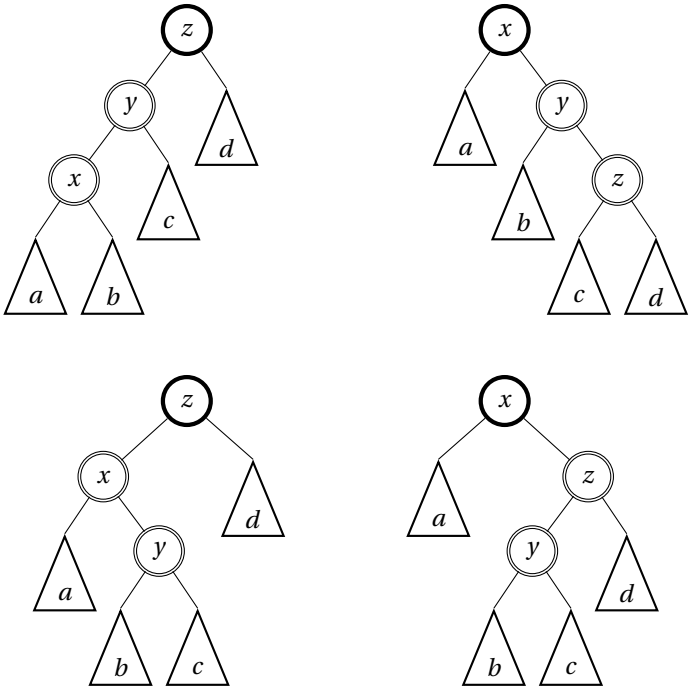
```
insert_leaf : 'a -> 'a rbt -> 'a rbt
```

implémentant l'insertion aux feuilles dans un arbre binaire de recherche.

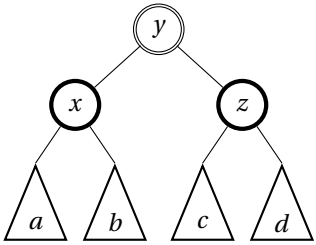
Question 7 Justifier que l'insertion aux feuilles créée **au plus** un conflit avec la règle (R_2).

Nous allons dans la suite analyser le conflit potentiellement créé, et voir comme le résoudre efficacement.

Les quatres formes de conflit possibles à une profondeur -1 sont représentés ci-dessous (les nœuds rouges étant représentés avec un double tour fin) :



À chaque fois, on peut le remplacer par :



Question 8

1. Justifier qu'en procédant à un tel échange, il reste au plus un conflit et que, s'il en subsiste un, sa profondeur a diminué strictement.
2. En itérant au besoin cette opération, justifier qu'à la fin, il reste au plus un conflit et que celui est à la racine.
3. Comment traîter l'éventuel conflit restant ?

Cela montrer qu'en effectuant une insertion aux feuilles et en traîtant l'éventuel conflit en le faisant remonter puis éventuellement disparaître, on obtient à la fin un arbre rouge-noir équilibré.

Question 9 Écrire une fonction

```
update_conflict : 'a rbt -> 'a rbt
```

qui prend en entrée un arbre, regarde si, à la racine, il est comme l'un de ceux dessinés plus haut (avec, donc, un conflit à une profondeur -1) et applique, au besoin, la transformation décrite.

Question 10 En déduire une fonction

```
insert : 'a -> 'a rbt -> 'a rbt
```

qui effectue une insertion suivie d'un rééquilibrage. Ainsi, si on a donné un argument un arbre rouge-noir équilibré, alors le résultat est aussi équilibré.

On s'assurera que la fonction `insert` est de complexité temporelle en $O(\log|t|)$.

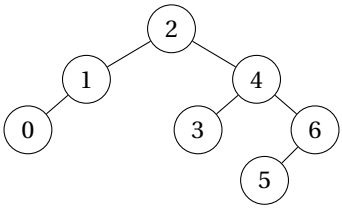
V. Suppression

Nous allons maintenant étudier l'autre opération de base des arbres binaires de recherche, la suppression d'un nœud.

1. Suppression « au-dessus des feuilles »

Dans cette section, on ne s'occupe que d'avoir des arbres binaires de recherche sans prendre en compte la coloration des nœuds.

Nous allons commencer par étudier une méthode de suppression pour les arbres binaires de recherche. Considérons l'arbre suivant :



Si l'on veut supprimer un nœud ayant une feuille comme fils gauche ou fils droite, c'est très simple (qu'obtient-on en supprimant la valeur 5 ? 1 ?). Si le nœud correspondant à la valeur à supprimer contient deux nœuds internes comme fils gauche et droit, on peut procéder ainsi :

1. on repère le minimum m du sous-arbre droit,
2. on supprime ce minimum (c'est facile, son sous-arbre gauche est une feuille),
3. on remplace la valeur à supprimer par m.

Question 11 Que se passe-t'il si l'on veut appliquer cette procédure à l'élimination du nœud 2 de l'arbre précédent ? du nœud 4 ?

Question 12 Écrire une fonction

```
delete : 'a -> 'a rbt -> 'a rbt
```

qui effectue la suppression d'une valeur d'un arbre binaire de recherche, en appliquant la méthode proposée.

2. Rééquilibrage après suppression

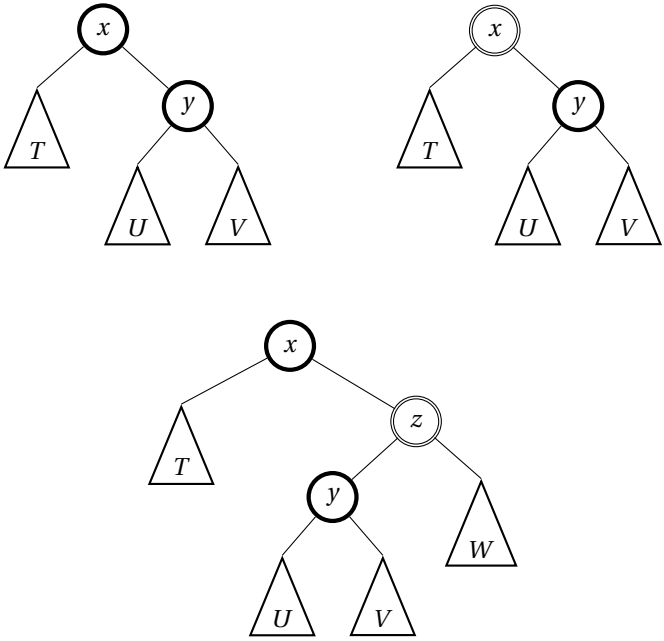
Nous allons maintenant étudier comment adapter cette méthode aux arbres rouges-noirs. Il est possible, suite à la suppression d'un nœud, qu'apparaissent un conflit avec la règle (R_2) et un déséquilibre violant la règle (R_3) : les hauteurs noires des sous-arbres gauche et droit d'un nœud peuvent être différentes.

Nous allons pour l'instant nous concentrer sur les violations de la règle (R_3).

Question 13 On suppose que, suite à la suppression d'un élément dans le sous-arbre gauche, celui-ci voit sa hauteur noire diminuer de 1, créant un déséquilibre

avec le sous-arbre droit. On suppose de plus que la racine du sous-arbre gauche est noire.

1. Justifier que l'on est nécessairement dans l'un des trois cas suivants ;
2. Pour chaque cas, proposer une transformation pour rééquilibrer l'arbre.
3. Dans chaque cas, indiquer si la hauteur noire de l'arbre en entier a diminué, et si il faut éventuellement procéder à une résolution de conflit vis-à-vis de la règle (R_2).



Question 14 Au départ, lorsque l'on supprime effectivement un nœud au-dessus d'une feuille, quels sont les différents cas possibles ? Lesquels peut engendrer un déséquilibre ?

Question 15 En déduire une fonction

```
delete2 : 'a -> 'a rbt -> 'a rbt
```

qui effectue la suppression de la valeur indiquée, rééquilibre et résout l'arbre obtenu, et renvoie l'arbre rouge-noir équilibré obtenu.

VI. Questions subsidiaires

L'écriture des différentes procédures de résolution de conflit et de rééquilibrage peuvent être à l'origine de nombreuses petites erreurs. Il convient donc de tester les fonctions obtenus.

Question 16 Décrire et programmer différentes procédures de test pour vous assurer que les fonctions écrites précédemment sont correctes (du moins, avec une bonne probabilité).