La structure d'arbre

A - Fonctions élémentaires de traitement des arbres

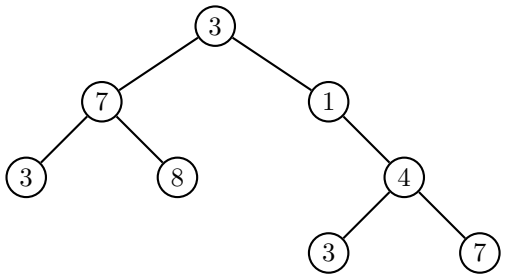
1) On utilise la structure :

```
type 'a arbre = Vide | Noeud of 'a arbre * 'a * 'a arbre;;
```

a) Écrire des fonctions renvoyant respectivement la valeur contenue à la racine, le fils gauche, le fils droit, la taille et la hauteur d'un arbre.

b) Écrire une fonction `recherche: 'a -> 'a arbre -> bool` qui teste si un élément `x` est présent dans un arbre `a`. Modifier cette procédure pour qu'elle renvoie un sous-arbre de `a` enraciné en `x` (et l'arbre vide si `x` n'est pas une clé de `a`).

c) Si `x` est une clé de l'arbre `a`, les mots associés à `(x,a)` sont les mots sur l'alphabet $\{g, d\}$ qui codent les chemins qui vont de la racine de `a` à un nœud d'étiquette `x`. Ainsi, dans l'arbre :



l'élément 3 est associés aux mots "", "gg" et "ddg". Écrire une fonction qui, appliquée à un arbre `a` et à un élément `x`, renvoie un message d'erreur si `x` n'est pas une clé de `a`, ou un mot associé à `x` sinon. Écrire une fonction qui, appliquée à un arbre `A` et à un élément `x`, renvoie la liste des mots associés à `x`.

2) On utilise la structure :

```
type arbre = Feuille of int | Noeud of arbre * arbre;;
```

a) Écrire une fonction qui, appliquée à un entier naturel non nul n, renvoie un arbre à n feuilles en utilisant la méthode suivante :

- si $n = 1$, on renvoie une feuille dont le contenu est un entier choisi aléatoirement ;
- sinon, on choisit k entre 1 et $n - 1$ et on renvoie un arbre dont les fils sont calculés récursivement, le fils gauche contenant k feuilles et le fils droit $n - k$ feuilles.

b) Écrire une fonction qui, appliquée à un entier naturel non nul n, renvoie un arbre peigne à n feuilles (un arbre peigne est soit réduite à une feuille, soit un arbre dont le fils gauche est un arbre peigne et le fils droit réduit à une feuille).

c) Écrire une fonction `copier` qui, appliquée à un arbre `a` et à un vecteur d'entier `V`, copie les contenus des feuilles de `a` dans les cases de `V` ; les feuilles seront parcourues de gauche à droite et le vecteur rempli depuis la cases $V.(0)$. On supposera que le vecteur est de longueur suffisante.

d) Écrire une fonction `liste_feuilles` qui, appliquée à un arbre `a`, renvoie la liste des contenus des feuilles de `a` lues de gauche à droite (la feuille la plus à gauche sera en tête de liste). Le temps de calcul de la fonction devra être de l'ordre de la taille de l'arbre.

3) les arbres par tableaux de fils ou de pères.

On définit des arbres (non nécessairement binaires) sur l'ensemble de sommets $S = \{0, 1, \ldots, n-1\}$. Par convention, la racine des arbres est toujours le sommet 0 et un tel arbre est défini ou bien par le `vecteur des fils`, noté `fils`, où `fils.(i)` est la liste des fils du sommets i, ou bien par le vecteur des pères noté `pere`, où `pere.(i)` est le numéro du père de i (par convention, `pere.(0)` vaut 0).

a) Écrire une fonction `PereEnFils: int vect -> int list vect` (resp. `FilsEnPere: int list vect -> int vect`) qui calcule le vecteur `fils` (resp. `pere`) en fonction du vecteur `pere` (resp. `fils`).

b) Le poids p_i d'un nœud i est le nombre de sommets du sous-arbre de sommet i. Écrire une fonction `poids: int list vect -> int vect` qui, appliqué au vecteur `fils` définissant un arbre, renvoie le vecteur $[|p_0; p_1; \ldots; p_{n-1}|]$. On ne fera qu'un parcours de listes de fils.

c) Un arbre est dit `gauche` si pour chaque nœud qui n'est pas une feuille, le premier de ses fils est son fils le plus lourd. Écrire une fonction `gauchir: int list vect -> int list vect` qui transforme un arbre quelconque en un arbre gauche équivalent (i.e. tel que les deux arbres aient le même vecteur `pere`).

B - Arbres binaires de recherche

4) Primitives de la structure d'ABR

On considère dans cet exercice la structure d'ABR définie par le type :

```
type abr = Vide | Noeud of abr * int * abr;;
```

Si a est un ABR, on note $\mathcal{C}(a)$ l'ensemble des clés stockées dans a. Nous supposerons que les nœuds de a contiennent des clés deux à deux distinctes.

a) Écrire une fonction `validation: abr -> bool` qui teste si un arbre a bien la structure d'arbre binaire de recherche. On veillera à ne parcourir l'arbre qu'une seule fois.

b) Écrire une fonction `lecture_infixe` qui, appliquée à un ABR a, renvoie la liste des valeurs stockées dans a obtenue par parcours infixé.

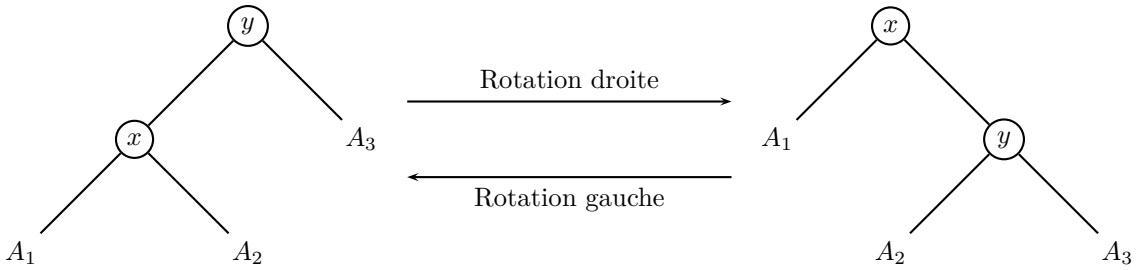
c) Écrire des fonctions `maximum` et `minimum` qui calculent respectivement le maximum et le minimum des clés stockées dans un ABR.

d) Écrire une fonction `inserer: int -> abr -> abr` permettant d'ajouter une clé à un ABR.

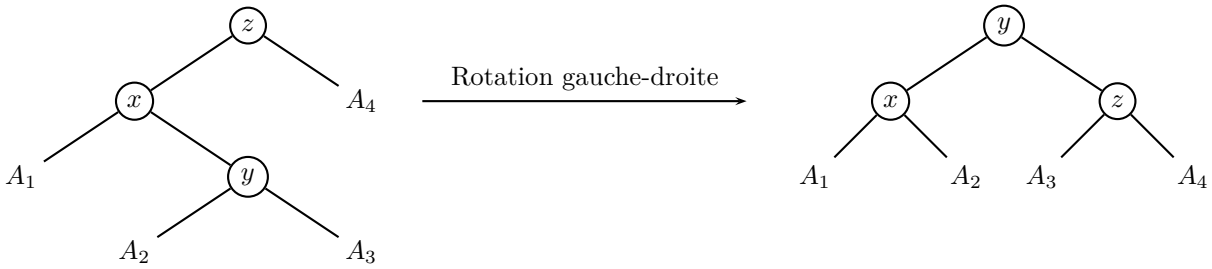
e) Écrire une fonction `supprimer_max: abr -> int*abr` (resp. `supprimer_min: abr -> int*abr`) qui, appliquée à un ABR non vide a, renvoit un couple (x, b) où $x = \max \mathcal{C}(a)$ (resp. $x = \min \mathcal{C}(a)$) et où b est un ABR tel que $\mathcal{C}(b) = \mathcal{C}(a) \setminus \{M\}$.

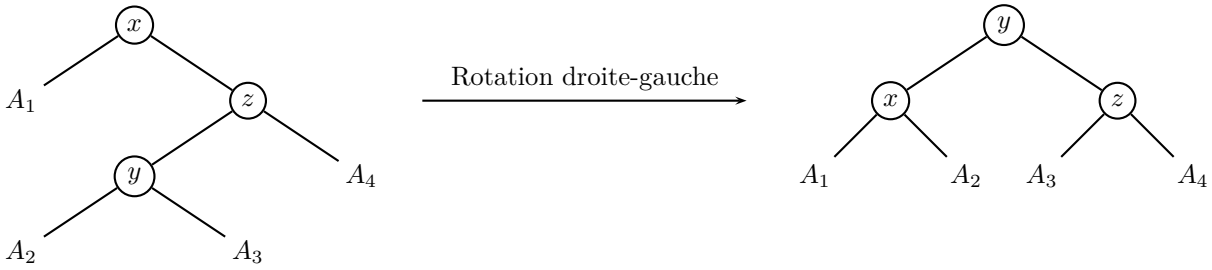
f) Écrire une fonction `supprimer_racine: abr -> abr` qui, appliquée à `Noeud`(fg, x, fd), renvoie un couple (x, b) où b est un ABR b tel que $\mathcal{C}(b) = \mathcal{C}(fg) \cup \mathcal{C}(fd)$.

g) Écrire une fonction qui, appliquée à un ABR, renvoient l'ABR obtenu après rotation droite ; si la rotation droite n'est pas permise, un message d'erreur sera renvoyé. Écrire la fonction symétrique qui effectue une rotation gauche.



h) Écrire une fonction qui, appliquée à un ABR, renvoient l'ABR obtenu après rotation gauche-droite ; si cette rotation n'est pas permise, un message d'erreur sera renvoyé. Écrire la fonction symétrique qui effectue une rotation droite-gauche.





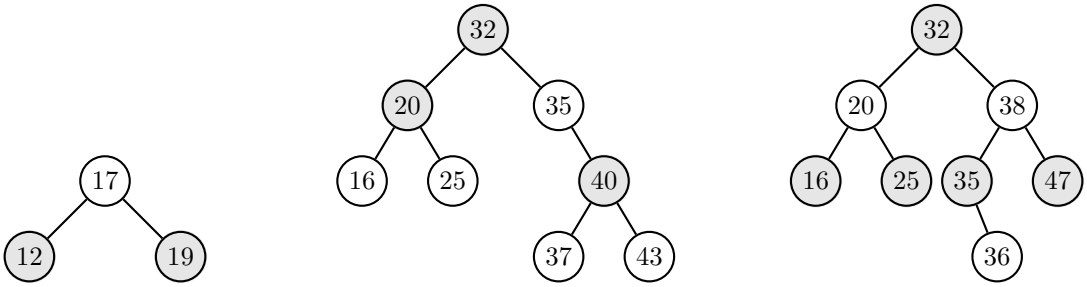
5) Arbres rouge-noir ... ou blanc-gris

Un arbre coloré est un arbre binaire de recherche dont les nœuds ont une étiquette "couleur", qui peut prendre la valeur `Blanc` ou la valeur `Gris`. Un arbre coloré à clés entières pourra donc être modélisé par la structure :

```
couleur = Blanc | Gris ;;
```

```
type arbre = V | N of couleur*arbre*int*arbre ;;
```

Les feuilles d'un arbre seront les sous-arbres vides qu'il contient. Par convention, une feuille n'a pas de couleur. Ainsi, les arbres $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ ci-dessous contiennent respectivement 4, 9 et 9 feuilles.



$\mathcal{A}_1$ $\mathcal{A}_2$ $\mathcal{A}_3$

Pour préserver l'équilibre des arbres, nous imposons les conditions suivantes :

(P1) pour chaque nœud n de l'arbre, chaque chemin qui part de n et qui arrive à une feuille contient le même nombre de nœuds gris. Ce nombre, noté $hg(n)$, est la *hauteur grise* de n ;

(P2) chaque nœud blanc possède ou bien deux fils vides, ou bien deux fils gris.

Un arbre coloré vérifiant ces propriétés sera appelé *arbre blanc-gris*, ou *ABG* en abrégé. La hauteur grise d'un ABG est la hauteur grise de sa racine. La hauteur d'un arbre est le nombre de nœud de sa branche la plus longue. Les arbres $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ sont de hauteurs respectives 2, 4 et 4. La taille d'un arbre est le nombre de ses nœuds (sans compter les feuilles vides). Les arbres $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ sont de tailles respectives 3, 8 et 8. Nous noterons respectivement $hg(\mathcal{A})$, $h(\mathcal{A})$ et $n(\mathcal{A})$ la hauteur grise, la hauteur et la taille d'un AGB $\mathcal{A}$.

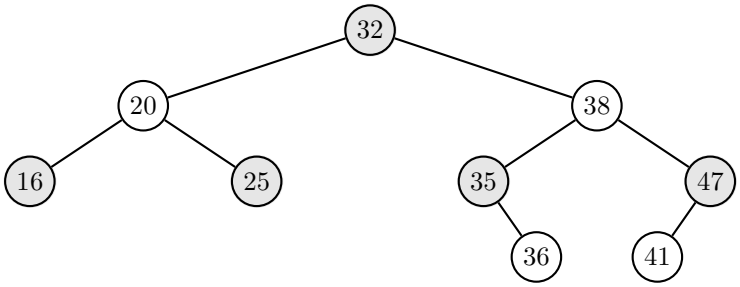
a) Les arbres $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ sont-ils des ABG ?

b) Pour tout AGB $\mathcal{A}$, montrer les inégalités :

$$hg(\mathcal{A}) \leq h(\mathcal{A}) \leq 2 \times hg(\mathcal{A}) \text{ et } 2^{hg(\mathcal{A})} \leq n + 1.$$

Commentez ces résultats.

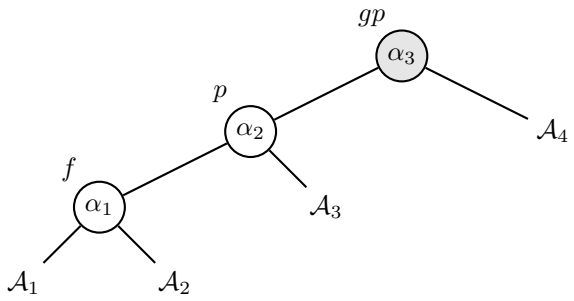
c) On souhaite insérer une nouvelle clé α dans un ABG $\mathcal{A}$. Nous commençons donc par effectuer une insertion comme dans un ABR classique, en créant un nouveau nœud, qui remplace une feuille ; ce nouveau nœud est coloré en blanc. Ainsi, l'insertion de la clé 41 dans l'arbre $\mathcal{A}_3$ donne l'arbre :



Montrer que la propriété (P1) est préservée par cette insertion ; dans quel cas la propriété (P2) ne l'est -elle pas ?

d) On suppose qu'un arbre coloré $\mathcal{A}$ vérifie (P1) mais qu'il bafoue la règle (P2) en un unique nœud p, qui est blanc et qui possède un fils blanc f (le second fils étant soit vide, soit noir).

- si p est la racine de $\mathcal{A}$, comment peut-on modifier l'arbre pour le changer en ABG ?
- on suppose maintenant que p est le fils de gp (qui est donc un nœud gris) et que nous sommes dans la situation suivante :



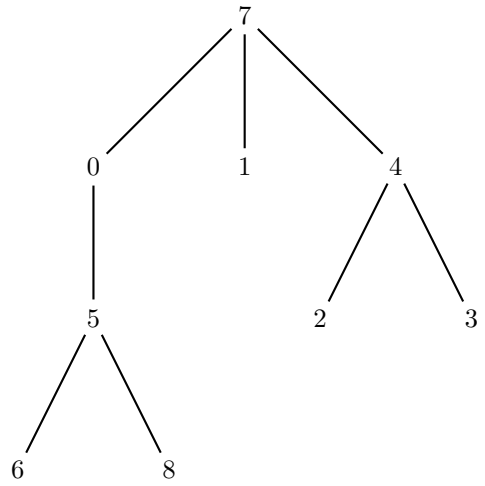
  Montrer que les quatre ABG $\mathcal{A}_i$ ont même hauteur grise ; en déduire une manière de modifier l'arbre enraciné en gp pour lui donner une structure d'ABG (on étudiera également le cas où $\mathcal{A}_3$ est vide). Montrer que les autres situations se traitent de la même manière.

e) Comment les résultats précédents permettent-ils d'insérer une nouvelle clé dans un ABG en préservant la structure d'ABG ? Appliquer votre méthode à l'ABG $\mathcal{A}_3$, en insérant tour à tour les valeurs 41, 43, 33, 23 et 24.

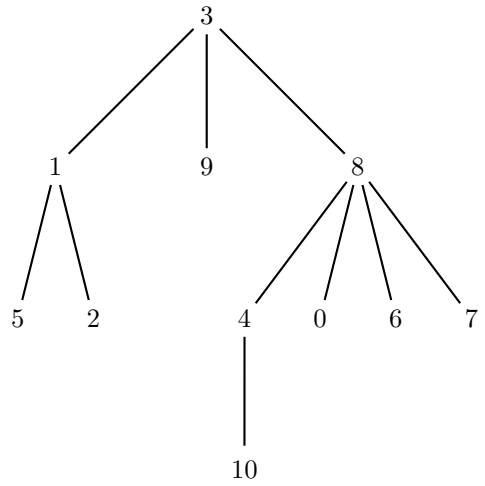
f) Écrire une fonction qui insère une clé dans un ABG.

Arbres non binaires

6) On considère des arbres dont les nœuds sont numérotés de 0 à $n-1$. Chaque nœud peut avoir un nombre de fils quelconque, comme dans les exemples ci-dessous :



arbre $\mathcal{A}_1$



arbre $\mathcal{A}_2$

Un tel arbre sera représenté par deux vecteurs de longueur n, notés `FilsGauche` et `FrereDroit`, avec les conventions suivantes : pour i compris entre 0 et $n-1$, `FilsGauche(i)` contient la valeur -1 si i est une feuille, et le numéro du fils de i situé le plus à gauche sinon ; `FrereDroit.(i)` contient -1 si i n'a pas de frère, et le numéro de son frère droit sinon. Ainsi, les arbres $\mathcal{A}_1$ et $\mathcal{A}_2$ sont représentés par les vecteurs :

$$\begin{cases} \texttt{FilsGauche}_1 = [|5;-1;-1;-1;2;6;-1;0;-1|] \\ \texttt{FrereDroit}_1 = [|1;4;3;-1;-1;-1;8;-1;-1|] \\ \texttt{FilsGauche}_2 = [|-1;5;-1;1;10;-1;-1;-1;4;-1;-1|] \\ \texttt{FrereDroit}_2 = [|6;9;-1;-1;0;2;7;-1;-1;8;-1|] \end{cases}$$

a) Écrire une fonction qui, en fonction des deux vecteurs `FilsGauche` et `FrereDroit` associé à un arbre, calcule le numéro de la racine de l'arbre.

b) On peut également associer à un arbre les vecteurs `pere` et `arite`, également de taille n : pour tout i compris entre 0 et $n-1$, `pere.(i)` contient la valeur -1 si i est la racine, et le numéro du père de i sinon ; `arite.(i)` contient le nombre de fils du nœud i.

Écrire une fonction qui calcule, en temps $O(n)$, le vecteur `arite` en fonction des vecteurs `FilsGauche` et `FrereDroit`

Écrire une fonction qui calcule, toujours en temps linéaire, le vecteur `arite` en fonction des vecteurs `FilsGauche` et `FrereDroit` (on remplira la table `pere` récursivement en partant de la racine de l'arbre).

7) On considère dans cet exercice des forêts d'arbres quelconques contenant des clés de type `int` : une forêt est une famille d'arbres $(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k)$ où k est un entier naturel (si $k = 0$, la forêt est dite vide) et où chaque $\mathcal{A}_i$ est un arbre non vide (l'arbre vide n'aura pas d'intérêt, puisque la forêt vide permet de traiter le cas où l'ensemble des données est vide).

a) Montrer que la structure utilisée habituellement pour définir les arbres binaires permet de manipuler les forêts d'arbre quelconques :

```
type foret = Vide | Noeud of foret*int*foret;;
```

Le nœud $N = \texttt{noeud}(Fils, a, Freres)$ sera le nœud contenant la clé a et ayant la forêt $Fils$ pour ensemble de fils et la forêt $Freres$ pour ensemble de frères "situés à droite".

b) Ecrire des fonctions qui, appliquées à une forêt, renvoient respectivement le nombre k d'arbres formant la forêt, la hauteur maximale d'un arbre de la forêt et le nombre total de nœuds de la forêt.