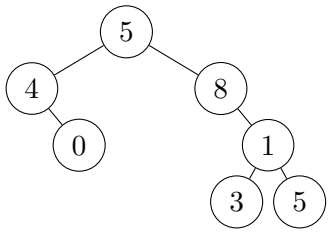
Codage d'arbres binaires

On définit le type d'arbres étiquetés :

```
type 'a abin = Vide | N of 'a abin * 'a * 'a abin;;
```

1) Grâce à ce type définir l'arbre `abin_exemple : int abin` qui correspond à l'arbre :



   Cet arbre servira à tester les fonctions des questions suivantes.

2) On rappelle qu'une *feuille* est un nœud (non vide) dont les deux fils sont vides ; la *taille* est le nombre de nœuds non vides ; et la `hauteur` est la longeur du plus long chemin entre la racine et une feuille.

   Écrire les fonctions : `nb_feuilles : 'a abin -> int`, `taille : 'a abin -> int` et `hauteur : 'a abin -> int`.

   Vérifier que `abin_exemple` possède 3 feuilles, est de taille 7 et de hauteur 3.

3) Écrire la fonction de parcours infixe qui retourne la liste des étiquettes :

   ```
   parcours_infixe : 'a abin -> 'a list
   ```

   Par exemple `parcours_infixe abin_exemple` doit retourner `[4, 0, 5, 8, 3, 1, 5]`

   On pourra utiliser l'opération de concaténation de listes `@` fournie par Caml.

   Même question avec le parcours préfixe.

   Même question avec le parcours postfixe.

4) En utilisant la fonction `parcours_infixe` écrire une fonction `est_ABR` qui teste si un arbre binaire dont les étiquettes sont des entiers est un arbre binaire de recherche.

5) Pour un arbre binaire d'entiers, écrire une fonction `somme : int abin -> int = <fun>` qui calcule la somme des étiquettes de l'arbre.

   Vérifier que `somme abin_exemple` retourne bien 26.

6) Écrire une fonction d'appartenance : `appartient : 'a abin * 'a -> bool` qui prend en argument un arbre binaire arbre et un objet `b` de type `'a`, et qui renvoie `true` si `b` est l'étiquette d'un des noeuds de l'arbre, `false` sinon.

   Tester avec l'arbre `abin_exemple`.

7) Modifier la fonction précédente en une fonction : `appartient_et_compte : 'a abin * 'a -> bool * int` telle que `appartient_et_compte x a` renvoie un couple constitué du booléen défini dans la question précédente et du nombre d'occurences de l'étiquette `x` dans l'arbre `a`.

   Tester différentes valeurs avec l'arbre `abin_exemple`.

8) Écrire une fonction : `parents : 'a abin -> 'a -> 'a list` qui pour un arbre binaire `arbre` et un objet `b` de type `'a` renvoie la liste des étiquettes des noeuds dont l'un des deux fils a pour étiquette `b`.

Codage d'arbre par un tableau

Dans cette section on étudie un type d'arbres particuliers (non nécessairement binaires) dont les sommets sont exactement les éléments de $[\![0, n-1]\!]$ (où n est le nombre de sommets de l'arbre. Ainsi chaque élément de $[\![0, n-1]\!]$ apparaît exactement une fois dans l'arbre.
On code l'arbre dans un tableau. Il y a deux types de codages possibles :

- Ou bien on code le tableau des pères : pour chaque entier $i \in [\![0, n-1]\!]$ la i–ième case du tableau est son père. Ainsi l'arbre est codé par un tableau de type `int array`. Par convention le père de la racine est codé par -1.
- Ou bien on code le tableau des fils : pour chaque entier $i \in [\![0, n-1]\!]$ la i–ième case du tableau est la liste de ses fils. Ainsi l'arbre est codé par un tableau de type `int list array`.

Exemple : considérons l'arbre :

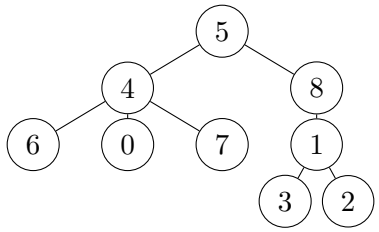


Tableau des pères :

0	1	2	3	4	5	6	7	8
4	8	1	1	5	-1	4	4	5

Codage sous Caml :
`[|4;8;1;1;5;-1;4;4;5|]`

Tableau des fils :

0	1	2	3	4	5	6	7	8
[ ]	[3; 2]	[ ]	[ ]	[6; 0; 7]	[4; 8]	[ ]	[ ]	[1]

Codage sous Caml :
`[|[];[3;2];[];[];[6;0;7];[4;8];[];[];[1]|]`

9) Définir en variable globale les deux tableaux : `exemple_peres` et `exemple_fils` qui correspondent aux deux codages du tableau donné en exemple ci–dessus. Ces deux tableaux serviront à tester des fonctions.

10) Écrire une fonction : `racine : int array -> int` dont le paramètre est tableau de pères et qui calcule la racine.

11) Écrire une fonction : `hauteur_par_peres : int array -> int -> int` telle que si `t` est le tableau des pères et si `i` est le numéro d'un sommet, alors `hauteur_par_peres t i` permet de calculer la hauteur du sommet `i` dans l'arbre.

12) Écrire une fonction : `peres2fils : int list array -> int array` qui permet de passer du codage par tableau des fils au codage par tableau des pères.

13) Écrire une fonction de conversion : `abin2fils : int abin -> int list array` qui permet à partir d'un arbre binaire de type `int abin` de générer le codage par tableau de fils.

On revient au codage d'arbre binaire par le type : `type 'a abin = Vide | N of 'a abin * 'a * 'a abin;;`

14) Le parcours en largeur est la liste des étiquettes où les nœuds sont lus par ordre croissant de profondeur, puis, à profondeur égale, de gauche à droite. Par exemple, le parcours en largeur de l'arbre `abin_exemple` est la liste $[5; 4; 8; 0; 1; 3; 5]$

    Écrire la fonction `largeur : 'a abin -> 'a list` qui effectue le parcours en largeur d'un arbre binaire, et renvoie la liste des noeuds obtenue lors de ce parcours.