Un fichier `fenelon_tp13.zip` est à télécharger sur la page web du cours et à extraire sur votre ordinateur dans votre dossier de travail.

1 Parcours d'un arbre

Dans cette partie on va s'intéresser au problème du parcours d'un arbre étiqueté : c'est à dire la recherche d'un algorithme permettant d'énumérer la liste des étiquettes des noeuds une et une seule fois. Deux principales stratégies existent : les parcours en largeur et en profondeur.
Dans toute cette section, on considère le type d'arbres binaires suivant :

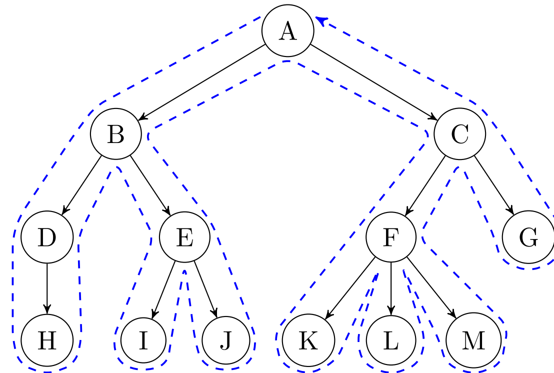
```
type 'a arbre = Vide | N of 'a * 'a arbre * 'a arbre
```

1. Écrire une fonction `nombre_noeuds : 'a arbre -> int` renvoyant le nombre de nœuds d'un arbre.
2. Écrire une fonction `hauteur : 'a arbre -> int` calculant la hauteur d'un arbre.

Exemple
```
# nombre_noeuds arbre_test ;;
- : int = 10
# hauteur arbre_test ;;
- : int = 4
```

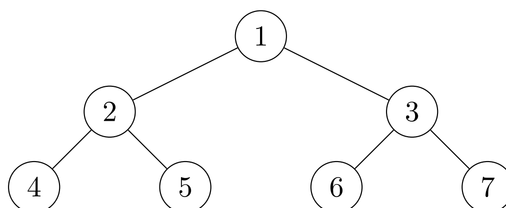
1.1 Parcours en profondeur

Le parcours en profondeur se fonde sur l'idée suivante : on va explorer l'arbre de manière récursive en parcourant d'abord l'intégralité du sous-arbre fils de gauche de la racine puis le sous-arbre droit.



Néanmoins, comme on le voit sur la figure plus haut, il faut convenir d'un "moment" où l'on énumère un noeud. On distingue alors :

- Le parcours préfixe : on visite le noeud lorsque l'on descend dans la branche, et qu'on le rencontre pour la première fois,
- Le parcours postfixe (ou suffixe) : on visite le noeud lorsque l'on remonte de la branche et qu'on le rencontre pour la dernière fois,
- Le parcours infixe : on visite chaque noeud ayant un fils gauche la seconde fois qu'on le rencontre et chaque noeud sans fils gauche à la descente (l'idée est de parcourir le noeud entre la fin de parcours du sous arbre gauche de ses descendants et le sous arbre droit) .



Par exemple, sur l'arbre complet ci-dessus, on obtient les énumérations suivantes :

- Parcours préfixe : $[1; 2; 4; 5; 3; 6; 7]$;
- Parcours postfixe : $[4; 5; 2; 6; 7; 3; 1]$;
- Parcours infixe : $[4; 2; 5; 1; 6; 3; 7]$.

3. Définir une variable globale `arbre_ex` représentant l'arbre ci-dessus.
4. Écrire les fonctions suivantes permettant d'énumérer les étiquettes d'un arbre en appliquant les trois algorithmes proposés ci-dessus :
   (a) `enum_prefixe : 'a arbre -> 'a list`
   (b) `enum_postfixe : 'a arbre -> 'a list`
   (c) `enum_infixe : 'a arbre -> 'a list`

   Indication : On pourra utiliser l'opérateur `@` pour concaténer des listes.

Exemple
```
# enum_prefixe arbre_ex ;;
- : int list = [1; 2; 4; 5; 3; 6; 7]
# enum_postfixe arbre_ex ;;
- : int list = [4; 5; 2; 6; 7; 3; 1]
# enum_infixe arbre_ex ;;
- : int list = [4; 2; 5; 1; 6; 3; 7]
```

1.2 Parcours en largeur

L'idée du parcours en largeur est d'explorer tous les noeuds à profondeur 0, puis tous ceux à profondeur 1, puis ceux à profondeur 2, etc. Sur l'arbre précédent, le parcours en largeur donne donc l'énumération : $[1; 2; 3; 4; 5; 6; 7]$.
L'implémentation est plus délicate que celle des parcours en profondeur. Dans cet exercice, nous allons recourir à la notion de forêt. On appelle forêt une liste d'arbres binaires.

5. Écrire une fonction `racines : 'a arbre list -> 'a liste` donnant la liste des racines des arbres constituant une forêt.
6. Écrire une fonction `ss_arbres : 'a arbre list -> 'a arbre list` renvoyant la liste des sous-arbres gauche et droits des arbres de la forêt.

On peut ensuite implémenter le parcours en largeur de la façon suivante :

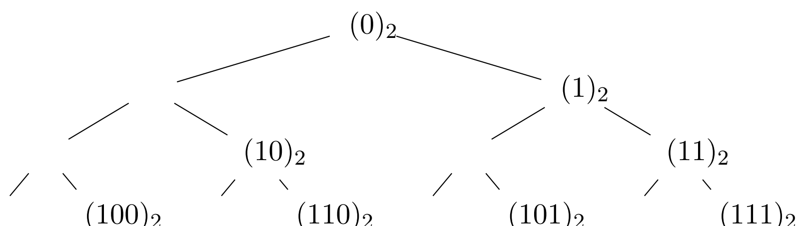
- on crée une liste contentant uniquement l'arbre en question (cela donne une première forêt) ;
- on extrait la liste des racines de cette forêt ;
- on crée une nouvelle forêt composée des arbres descendants des racines de cette forêt (notez que l'on "descend d'un étage" avec cette opération) ;
- on applique récursivement la procédure.

7. Écrire une fonction `enum_largeur : 'a arbre -> 'a list` implémentant cet algorithme.

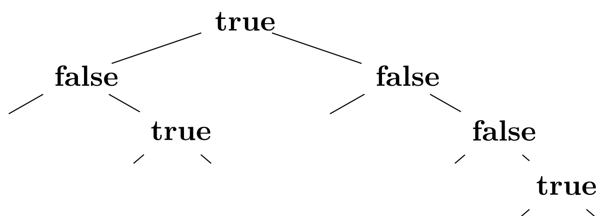
Exemple
```
# enum_largeur arbre_ex ;;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

2 Arbres radix

On se propose de décrire une structure de données pour représenter les ensembles finis d'entiers positifs ou nuls appelée arbres radix (radix tree en anglais). L'idée consiste en une structure arborescente basée sur l'écriture binaire de l'entier : si l'entier vaut 0 il est stocké à la racine ; sinon on le stocke dans le sous-arbre gauche si son dernier chiffre binaire est 0 (donc s'il est pair), et dans le sous-arbre droit si son dernier chiffre binaire est 1 (s'il est impair), et l'on poursuit avec les chiffres suivants. À titre indicatif, voici où se retrouvent placés les huit premiers entiers (leur écriture binaire est indiquée entre parenthèses) :



Notez que les chiffres de l'écriture binaire sont utilisés de droite à gauche (du moins significatif au plus significatif). On choisit de coder les arbres radix en OCaml par des arbres avec des feuilles vides et des noeuds contenant un booléen indiquant la présence ou l'absence de l'entier correspondant dans l'ensemble représenté par l'arbre. L'ensemble $\{0; 2; 7\}$ sera donc représenté par l'arbre suivant :



Par la suite, on conviendra d'identifier un ensemble d'entiers avec l'arbre radix qui le représente.
On définit donc le type :

```
type arbre = Vide | Noeud of (bool * arbre * arbre)
```

8. Écrire l'arbre représenté dans la figure précédente.
9. Écrire une fonction `cherche : int -> arbre -> bool` prenant en arguments un entier n et un ensemble E (représenté par un arbre radix) et qui renvoie un booléen déterminant l'appartenance de n à E.
10. Écrire une fonction `ajoute : int -> arbre -> arbre` prenant en arguments un entier n et un ensemble E et qui retourne l'ensemble $E \cup \{n\}$.
11. Écrire une fonction `construit : int list -> arbre` construisant un ensemble à partir de la liste des entiers qu'il contient.
12. Écrire une fonction `supprime : int -> arbre -> arbre` prenant en arguments un entier n et un ensemble E et retournant l'ensemble $E \setminus \{n\}$ (si n n'appartient pas à E, cette fonction ne doit pas échouer mais retourner le même ensemble E).
13. La fonction de la question précédente peut renvoyer un arbre avec des branches mortes (uniquement avec des noeuds à `false`). Proposer une fonction `elague : arbre -> arbre` prenant un arbre radix en entier et enlève les branches mortes.
14. Écrire une fonction `union : arbre -> arbre -> arbre` réalisant l'union de deux ensembles.
15. Écrire une fonction `intersection : arbre -> arbre -> arbre` réalisant l'intersection de deux ensembles.

3 Arbres quaternaires et compression d'images

On s'intéresse à des images carrées dont la taille est un multiple de 4. L'image est découpée en quatre sous-zones qui sont elles-mêmes découpées de manière récursive.

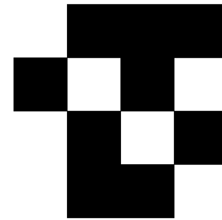
3.1 Arbres quaternaires

On considère le type suivant :

```
type quaternaire = Blanc
                 | Noir
                 | Noeud of quaternaire * quaternaire * quaternaire * quaternaire
```

À l'aide de ce type, l'image ci-contre peut être représentée par :

```
Noeud(
  Noeud(Blanc,Noir,Noir,Blanc),
  Noeud(Noir,Blanc,Noir,Blanc),
  Noeud(Noir,Blanc,Noir,Noir),
  Noeud(Noir,Blanc,Noir,Blanc)
)
```



16. Proposer un arbre quaternaire représentant un damier 4×4.
17. Écrire une fonction `damier : int -> quaternaire` renvoyant un arbre quaternaire représentant un damier de taille $2^n \times 2^n$. Votre fonction devra n'effectuer qu'un seul appel récursif.
18. Pour les dessins nous allons avoir besoin des commandes suivantes :

```
open Graphics (*Bibliothèque graphique*)
#load "graphics.cma"
open_graph "" d'une fenêtre vide*)
```

En utilisant la commande `fill_rect x y largeur` qui permet de remplir un carré de côté largeur dont le coin inférieur gauche est situé aux coordonnées (x, y), écrire une fonction `dessin : quaternaire -> unit` permettant de dessiner l'image représentée par un arbre quaternaire.

3.2 Compression d'images

Dans cette partie nous allons nous intéresser à des images binaires dont la taille n est un multiple de 4. Une image de taille $n \times n$, est un ensemble de pixels chacun possédant une couleur "Blanc" ou "Noir". Comme dans la partie précédente, une telle image peut être représentée par un arbre quaternaire possédant $n \times n$ feuilles.

Cependant, cette représentation n'est pas très économique (on considère que ce qui est coûteux c'est de stocker les feuilles de l'arbre) et on va essayer de l'améliorer.

19. On considère l'image 16×16 représentée par l'arbre quaternaire ci-contre. Proposer un arbre quaternaire plus économique en mémoire consommée qui représente la même image.
20. En généralisant ce procédé, écrire une fonction `compression : quaternaire -> quaternaire` prenant en argument un arbre quaternaire et renvoyant sa version la plus compressée possible.

```
Noeud(
  Noeud(Blanc,Blanc,Blanc,Blanc),
  Noeud(Noir,Blanc,Noir,Noir),
  Noeud(Blanc,Blanc,Noir,Noir),
  Noeud(Noir,Noir,Noir,Noir)
)
```