1 Travail à faire

1.1 Files

On rappelle qu'une file (en anglais *queue* ou *FIFO, First In First Out*) est une structure de données abstraite offrant les fonctionnalités suivantes :

- tester si la file est vide ;
- enfiler un élément (en fin de file) ;
- défiler un élément de la file (élément de tête).

Implémentation avec deux listes

Dans ce problème, l'implémentation des files d'attente est effectuée par des couples de listes :

```
type 'a file = 'a list * 'a list;;
```

Soit une file `(arrive,repart) : 'a list * 'a list`. Pour implémenter les fonctionnalités de file on procède de la manière suivante :

- Pour tester si la file est vide, on teste si les listes `arrive` et `repart` sont vides, ce qui se fait en complexité $O(1)$.
- Pour enfiler un élément x, on ajoute x en tête de `arrive`, ce qui se fait en complexité $O(1)$.
- Pour défiler un élément de la file :
  - Si `repart` n'est pas vide, on se contente de retirer l'élément de tête de `repart` ce qui se fait en complexité $O(1)$.
  - Si `repart` est vide, on remplace `repart` par le miroir de `arrive`, on remplace `arrive` par la liste vide, et on retire l'élément de tête de la (nouvelle) liste `repart`.

Exemple de succession d'opérations :

Opération effectuée	Liste `arrive`	Liste `repart`
	[]	[]
enfile 3	[3]	[]
enfile 5	[5;3]	[]
enfile 4	[4;5;3]	[]
defile $\rightarrow$ 3	[]	[5;4]
enfile 8	[8]	[5;4]
defile $\rightarrow$ 5	[8]	[4]
defile $\rightarrow$ 4	[8]	[]
defile $\rightarrow$ 8	[]	[]

Boîte à outils

Écriture de quelques fonctions utiles pour la suite. Pensez à tester vos fonctions !

1) Écrire une fonction Caml récursive `taille` retournant la taille d'une liste.

```
taille :  'a list -> int
```

2) Écrire une fonction Caml `miroir`, de paramètre une liste $[a_0; a_1; \ldots; a_{n-1}]$, qui renvoie la liste $[a_{n-1}; \ldots; a_1; a_0]$ *en complexité linéaire.*

```
miroir :  'a list -> 'a list
```

(Une fonction auxiliaire pourra éventuellement être utilisée)

Fonctionnalités de base

Ces fonctions peuvent utiliser l'implémentation du type file.

3) Écrire une fonction Caml :

```
new_file : unit -> 'a file
```

permettant de créer une file initialement vide.

Remarque sur les fonctions sans paramètre : il y a un seul paramètre que est `()` de type `unit`. Ainsi le début de la déclaration de cette fonction sera `let new_file () = ...`

Si on veut forcer le type de la valeur de retour on peut même écrire

```
let new_file () : 'a file = ...
```

ce qui permet bien de créer une fonction de type `unit -> 'a file`

(au lieu de `unit -> 'a list * 'b list`)

4) Écrire une fonction Caml

```
est_vide : 'a file -> bool
```

permettant de tester si une file est vide.

5) Écrire une fonction Caml

```
enfile : 'a -> 'a file -> 'a file
```

de paramètres un élément e et une file f, permettant d'ajouter e en queue de la file f.

Pour forcer le second paramètre à être de type `'a file` (et pas `'a list * 'b list`) on commencera la déclaration de la fonction par :

```
let enfile a (f : 'a file) : 'a file = ....
```

6) Écrire la fonction `defile` qui renvoie un couple formé de l'élément défilé et de la nouvelle file :

```
defile : 'a file -> 'a * 'a file
```

7) Tester vos fonctions `new_file`, `est_vide`, `enfile` et `defile`. On pourra par exemple effectuer les opérations données dans l'en–tête du sujet :

```
let f1 = enfile 4 (enfile 5 (enfile 3 (new_file ())));;
let (x1, f2) = defile f1;;
let (x2, f3) = defile (enfile 8 f2);;
let (x3, f4) = defile f3;;
let (x4, f5) = defile f4;;
```

On doit trouver :

x1	int = 3
x2	int = 5
x3	int = 4
x4	int = 8
f1	int file = ([4; 5; 3], [])
f2	int file = ([], [5; 4])
f3	int file = ([8], [4])
f4	int file = ([8], [])
f5	int file = ([], [])

8) Écrire une fonction `longueur` retournant la longueur d'une file.

```
longueur :  'a file -> int
```

9) Écrire une fonction `elements` qui renvoie les éléments d'une file sous forme d'une chaîne, séparés par le symbole >>> indiquant l'ordre d'insertion dans la file.

```
elements : int file -> string
```

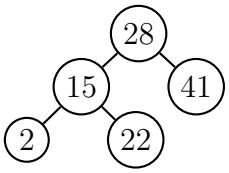
Avec l'exemple précédent, l'appel `elements f2` renverra la chaîne `"5>>>4"`.

1.2 Arbres binaires

Dans cette section nous allons utiliser le type d'arbre binaire étiqueté par le type `'a`. Chaque nœud est ou bien vide ou bien étiqueté, il possède alors deux fils (chacun étant vide ou non).

```
type 'a arbre = V | N of 'a arbre * 'a * 'a arbre;;
```

Par convention, pour chaque nœud non vide `N (g, e, d)`, `g` est le fils gauche et `d` est le fils droit (et `e` est l'étiquette).

10) Comment coder l'arbre 

?

Par la suite cet arbre sera nommé `arbre_ex` et servir à tester vos fonctions.

11) Écrire les fonctions de calcul de taille et de hauteur. Par convention, l'arbre vide est de hauteur -1 et de taille 0.

```
hauteur : 'a arbre -> int
taille : 'a arbre -> int
```

Test : `arbre_ex` est de hauteur 2 et de taille 5.

12) Calculer la somme des étiquettes d'un arbre :

```
somme : int arbre -> int
```

Test : la somme des étiquettes de `arbre_ex` est 108.

13) Écrire une fonction qui renvoie la liste des étiquettes d'un arbre (l'ordre ce ces étiquettes n'a pas d'importance, on ne cherchera pas à supprimer d'éventuels doublons).

```
liste_etiquettes : 'a arbre -> 'a list
```

Test : `liste_etiquettes arbre_ex` renvoie `[28; 15; 2; 22; 41]` (ou toute permutation de ces éléments).

14) On définit récursivement la notion de descendant : un descendant d'un nœud non vide n est ou bien n ou bien un descendant de son fils gauche ou bien un descendant de son fils droit.

On souhaite écrire une fonction qui, étant donné une étiquette e de type `'a` et un arbre, renvoie la liste des étiquettes de tous les descendants des nœuds ayant e comme étiquette.

```
descendants : 'a -> 'a arbre -> 'a list
```

Tests (l'ordre des descendants n'a pas d'importance) :

```
# descendants 15 arbre_ex;;
- : int list = [15; 2; 22]
# descendants 28 arbre_ex;;
- : int list = [28; 15; 2; 22; 41]
```

2 Pour aller plus loin, si vous avez le temps

2.1 Files de Hamming

Un entier de Hamming est un entier naturel non nul dont les seuls facteurs premiers éventuels sont 2, 3, 5.
Le problème de Hamming consiste à énumérer, dans l'ordre croissant, les n premiers nombres de Hamming.
Le premier entier de Hamming est 1 et tout autre entier de Hamming est le double, le triple ou le quintuple d'un entier de Hamming.
On utilise trois files $h2$, $h3$ et $h5$. Initialement ces trois files contiennent juste l'élément 1.
On calcule un à un les nombres de Hamming, dans l'ordre croissant, en calculant le plus petit entier e en tête des files $h2$, $h3$ et $h5$, en retirant e des files $h2$, $h3$ et $h5$ qui contiennent e, puis en enfilant respectivement $2e$, $3e$ et $5e$ aux files $h2$, $h3$ et $h5$.

15) Écrire une fonction Caml `hamming` qui renvoie les n premiers nombres de Hamming, par ordre croissant.

```
hamming : int -> int list
```

Indication : la fonction `min : 'a -> 'a -> 'a` permet de calculer le minimum de deux entiers.

Test : les 30 premiers nombres de Hamming sont 1; 2; 3; 4; 5; 6; 8; 9; 10; 12; 15; 16; 18; 20; 24; 25; 27; 30; 32; 36; 40; 45; 48; 50; 54; 60; 64; 72; 75; 80.

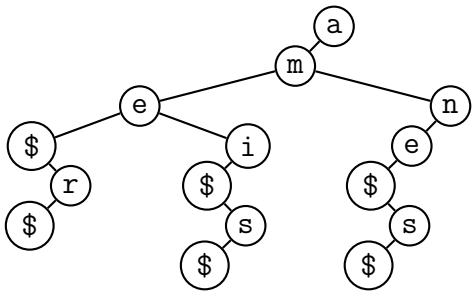
2.2 Arbre dictionnaire (inspiré du sujet X 2006)

On souhaite mémoriser des mots dans un arbre binaire étiqueté par des caractères :

```
type dict = char arbre;;
```

La recherche d'un mot se fait de façon suivante : on parcourt les lettres du mot une à une, en descendant systèmatiquement dans le fils gauche, tant que l'étiquette du nœud où on est correspond à la lettre qu'on est en train d'examiner. Si on arrive à un nœud dont l'étiquette ne correspond pas à la lettre du mot examinée, on cherche cette lettre dans son fils droit. Le caractère spécial `$` est un marqueur de fin de mot. Par convention le marqueur de fin de mot `$` ne peut être l'étiquette que d'un fils gauche, sinon on n'impose aucun ordre sur les lettres.
Par exemple, l'arbre suivant mémorise les mots :

ame
amer
ami
amis
ane
anes



Pour tests, cet arbre est codé par `N (N (N (N (V, '$', N (N (V, '$', V), 'r', V)), 'e', N (N (V, '$', N (N (V, '$', V), 's', V)), 'i', V)), 'm', N (N (N (V, '$', N (N (V, '$', V), 's', V)), 'e', V), 'n', V)), 'a', V)`

16) On souhaite implémenter les fonctionnalités suivantes (un mot sera donné par la liste de ses caractères) :

- Tester si un mot est dans un dictionnaire :

  `estDansDict : char list -> dict -> bool`

- Calculer la liste des mots du dictionnaire :

  `contenuDict : dict -> char list list`

- Ajouter un mot à un dictionnaire, en renvoyant le nouveau dictionnaire :

  `ajoutDict : char list -> dict -> dict`