Sauf mention contraire, le code est à écrire en OCaml.

On rappelle qu'un tas (binaire) max est un arbre binaire presque complet (tous les niveaux sont complets, sauf éventuellement le dernier) dont l'étiquette de chaque noeud est supérieure à ses éventuels fils.

I Un tas de questions

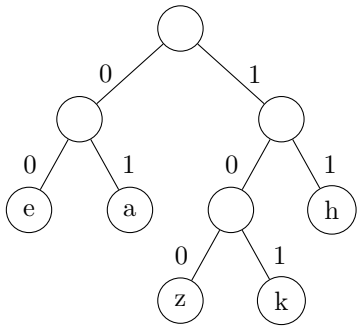
1. Dessiner le tas min (sous forme d'arbre et de tableau) obtenu à partir d'un tableau vide en ajoutant 3, 1, 2, 0 puis en supprimant deux fois le minimum puis en ajoutant 5, 0, 1, 4.
2. Dessiner le tas max (sous forme d'arbre et de tableau) obtenu en convertissant le tableau `[|3; 1; 0; 8; 4; 5; 6; 2; 7|]` en tas, en utilisant l'algorithme linéaire du cours.
3. Écrire une fonction en C pour vérifier qu'un tableau représente bien un tas max. Complexité?
4. Écrire une fonction `heap_to_tree : 'a heap -> 'a tree` pour convertir un tas (sous forme de tableau) en arbre, avec `type 'a heap = { a : 'a array; mutable n : int }` et `type 'a tree = E | N of 'a * 'a tree * 'a tree`.
5. Écrire une fonction `kfusion : 'a list list -> 'a list` en $O(n\log(k))$ pour fusionner k listes triées en une unique liste triée de taille n. On utilisera une FP min avec la fonction `create : unit -> 'a fp` et le type suivant :

```
type 'a fp = {
    add : 'a -> unit;
    is_empty : unit -> bool;
    take_min : unit -> 'a
}
```

   On pourra aussi utiliser la possibilité de comparer des listes avec `<` (dans l'ordre lexicographique).

II Compression de Huffman

La compression de Huffman consiste à coder chaque caractère d'un texte par une suite de 0 et de 1 d'autant plus courte que le caractère apparaît souvent. Ainsi, on espère réduire l'espace utilisé. Pour cela, à partir d'un tableau de couples (caractère, fréquence), on construit un arbre (de Huffman) dont les feuilles sont les caractères et chaque chemin de la racine à une feuille correspond à son codage. Dans l'arbre suivant, a est codé par [0; 1], z par [1; 0; 0]...



Comme un arbre de Huffman contient de l'information seulement aux feuilles, on utilise le type:

```
type 'a tree_huffman = F of 'a | N of 'a tree_huffman * 'a tree_huffman;;
```

1. Écrire une fonction `read t l` qui renvoie un couple constitué:
   - du premier caractère obtenu en se déplaçant, dans l'arbre de Huffman `t`, suivant les éléments (0 ou 1) de la liste `l`
   - de la partie de `l` non lue

   On supposera que `l` est une liste valide (qui correspond bien à une suite de caractères).
2. Écrire une fonction `decode : 'a tree_huffman -> int list -> 'a list` qui décode une liste de 0 et 1 avec un arbre de Huffman.

On veut maintenant construire l'arbre de Huffman. Pour cela on utilise une file de priorité min `fp` contenant des couples (fréquence, arbre), ordonné par fréquence croissante:

- Initialement, ajouter à `fp` tous les caractères (en tant que feuille) avec leur fréquence.
- Tant que possible: retirer les deux plus petits couples `(f1, a1)` et `(f2, a2)` de `fp` et y rajouter `(f1+f2, N(a1, a2))`.

On peut montrer que le dernier arbre obtenu est alors celui de Huffman (il minimise la longueur moyenne du codage d'un texte). On utilisera le type abstrait de file de priorité suivant:

```
type 'a fp = {
add : 'a -> unit;
is_empty : unit -> bool;
take_min : unit -> 'a
}
```

III Arbretas (en anglais: Treap = Tree + Heap)

On peut montrer qu'un arbre binaire de recherche (ABR) construit en ajoutant un à un n entiers choisis « uniformément au hasard » a une hauteur moyenne $O(\log(n))$.
Si les éléments à rajouter ne sont pas générés aléatoirement, mais sont tous connus à l'avance, on peut commencer par les mélanger aléatoirement puis les rajouter dans cet ordre aléatoire pour obtenir à nouveau une hauteur moyenne $O(\log(n))$. Pour cela, on considère l'algorithme suivant (mélange de Knuth), où `Random.int (i+1)` renvoie un entier uniformément au hasard entre 0 et `i`:

```
let shuffle t =
  for i = 0 to Array.length t - 1 do
    swap t i (Random.int (i+1))
  done;;
```

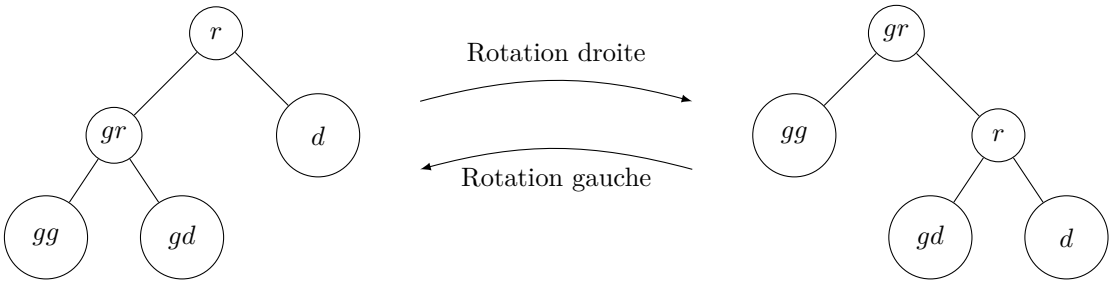
1. Écrire la fonction `swap : 'a array -> int -> int` utilisée par `shuffle`, telle que `swap t i j` échange `t.(i)` et `t.(j)`.
2. (Facultatif) Montrer que `shuffle t` applique une permutation choisie uniformément au hasard sur le tableau `t`.

Lorsque la totalité des éléments à rajouter n'est pas connue à l'avance, on peut utiliser une structure appelée **arbretas** qui est un arbre binaire (défini par `type 'a arb = V | N of 'a * 'a arb * 'a arb`) dont les noeuds sont étiquetés par des couples (élément, priorité), où la priorité est un nombre entier choisi uniformément au hasard au moment de l'ajout de l'élément. De plus:

- les éléments doivent vérifier la propriété d'ABR.
- la priorité d'un sommet doit être inférieure à la priorité de ses éventuels fils (propriété de tas min sur les priorités).

3. Dessiner un arbretas dont les couples (élément, priorité) sont: (1, 4), (5, 6), (3, 8), (2, 2), (0, 7).
4. Étant donnés des éléments et priorités tous distincts, montrer qu'il existe un unique arbretas les contenants.

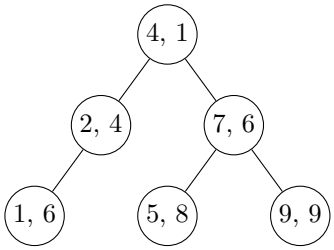
Nous allons utiliser des opérations de rotation sur un arbretas `N(r, N(gr, gg, gd), d)`:



5. Écrire une fonction `rotd` effectuant une rotation droite sur un arbre `N(r, N(gr, gg, gd), d)`.

On supposera définie dans la suite une fonction `rotg` pour effectuer « l'inverse » d'une rotation droite. On remarquera que, si `a` est un ABR, `rotg a` et `rotd a` sont aussi des ABR.
Pour ajouter un sommet s dans un arbretas (en conservant la structure d'arbretas), on l'ajoute comme dans un ABR classique (en ignorant les priorités) puis, si sa priorité est inférieure à celle de son père, on applique une rotation sur son père pour faire remonter s et on continue jusqu'à rétablir la structure d'arbretas.

6. Dessiner l'arbretas obtenu en rajoutant (6, 0) à l'arbretas suivant:



7. Écrire une fonction utilitaire `prio` renvoyant la priorité de la racine d'un arbre (on renverra `max_int`, c'est à dire le plus grand entier représentable en base 2 sur le processeur, si cet arbre est vide).
8. Écrire une fonction `add a e` ajoutant `e` (qui est un couple (élément, priorité)) à un arbretas `a`, en conservant la structure d'arbretas.

Pour supprimer un élément d'un arbretas, on commence par le chercher comme dans un ABR classique (en ignorant les priorités) puis on le fait descendre avec des rotations jusqu'à ce qu'il devienne une feuille qu'on peut alors supprimer librement.

9. Écrire une fonction `del` supprimant un élément dans un arbretas, en conservant la structure d'arbretas.

IV Conversion d'arbre binaire de recherche en tas

On définit un arbre binaire par: `type 'a arb = V | N of 'a * 'a arb * 'a arb;;`
On représente un tas par un tableau comme dans le cours.

1. Écrire une fonction `infixe : 'a arb -> 'a list`, si possible en complexité linéaire, renvoyant la liste des sommets d'un arbre parcourus dans l'ordre infixe.
2. Écrire une fonction `tas_of_abr : 'a arb -> 'a array` telle que, si `a` est un arbre binaire de recherche à n éléments, `tas_of_abr a` renvoie, en $O(n)$, un tableau représentant un tas min avec les mêmes éléments que `a`. On pourra utiliser `Array.of_list` qui convertit une `list` en `array`.
3. Écrire une fonction `abr_of_tas : 'a array -> 'a arb` en $O(n \log(n))$ telle que, si `t` représente un tas min à n éléments (et $n =$ `Array.length t`), `abr_of_tas t` renvoie un arbre binaire de recherche presque complet et avec les mêmes éléments que `t`. On pourra utiliser directement une fonction `take: 'a array -> 'a` pour renvoyer et supprimer la racine d'un tas, comme dans le cours.
4. Peut-on faire mieux que $O(n \log(n))$ pour la question précédente?