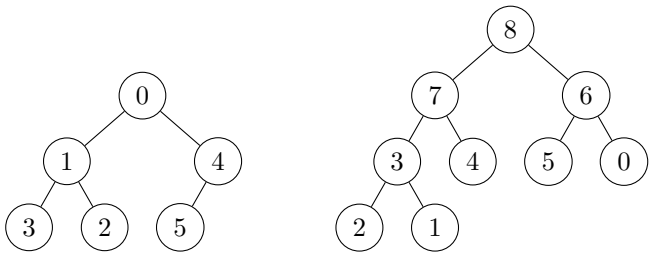
Sauf mention contraire, le code est à écrire en OCaml.

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.

> **Solution** : Vérifiez que vous obtenez le tas de gauche pour la question 1 et celui de droite pour la question 2.
>
> 
>

3. Écrire une fonction en C pour vérifier qu'un tableau représente bien un tas max. Complexité?

> **Solution** : On pourrait tester si `t[i] >= t[2*i+1]` (fils gauche) et `t[i] >= t[2*i+2]` (fils gauche) pour tout `i`, mais cela demande de faire attention à ne pas dépasser. Il est donc plus judicieux de tester si `t[i] <= t[(i - 1)/2]`, c'est-à-dire si chaque noeud est inférieur à son père (il faut juste faire attention à la racine qui n'a pas de père) :
>
> ```c
> bool is_heap(int* t, int n) {
>     for(int i = 1; i < n; i++)
>         if(t[i] > t[(i - 1)/2])
>             return false;
>     return true;
> }
> ```

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`.

> **Solution** :
>
> ```
> let rec heap_to_tree i =
>     if i >= heap.n then E
>     else N(heap.a.(i), heap_to_tree (2*i + 1), heap_to_tree (2*i + 2)) in
> ```

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 le type et la fonction de création suivant :

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

create : unit -> 'a fp (* créé un fp vide *)
```

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

> **Solution** : On insère d'abord toutes les listes dans la FP. Puis, tant que possible, extraire la tête de liste minimum

de la FP et l'ajouter à la liste de sortie. Il y a k listes dans la FP donc les opérations `add` et `take_min` sont en $O(\log(k))$, en utilisant une FP implémentée par tas. On effectue k `add` et n `take_min`, d'où la complexité $O(k\log(k)) + O(n\log(k)) = O(n\log(k))$ (en supposant que les listes sont non vides, on a $n \geq k$).

```
let kfusion ll =
    let f = create () in
    let rec init = function
        | [] -> ()
        | l::q -> f.add l; init q in
    init ll;
    let rec aux () = # pour construire la liste de retour
        match f.take_min () with
            | [] -> []
            | e::q -> f.add q; e::aux () in
    aux ()
```

2ème méthode (sans utiliser de FP...) : utiliser la même idée que le tri fusion. On sépare les listes à fusionner en deux, qu'on fusionne récursivement. On obtient ainsi deux listes triées qu'on fusionne ensuite avec la même fonction de fusion de liste (`list_fusion`) que le tri fusion.

```
let rec divise = function
    | [] -> [], []
    | e1::e2::q -> let l1, l2 = divise q in
                   e1::l1, e2::l2

let rec fusion = function (* fusion de 2 listes triées *)
    | [], l2 -> l2
    | l1, [] -> l1
    | e1::q1, e2::q2 -> if e1 < e2 then e1::fusion q1 (e2::q2)
                        else e2::fusion (e1::q1) q2

let rec kfusion = function (* fusion d'un nombre quelconque de listes triées *)
    | [l] -> l
    | ll -> let ll1, ll2 = divise ll in
            fusion (kfusion ll1) (kfusion ll2)
```

Autre possibilité, en fusionnant les listes 2 à 2 à chaque étape, jusqu'à obtenir une seule liste triée :

```
let rec fusion = function (* fusion de 2 listes triées *)
    | [], l2 -> l2
    | l1, [] -> l1
    | e1::q1, e2::q2 -> if e1 < e2 then e1::fusion q1 (e2::q2)
                        else e2::fusion (e1::q1) q2

let rec kfusion = function (* fusion d'un nombre quelconque de listes triées *)
    | [l] -> l
    | ll -> let rec fusion_pair = function
                | l1::l2::q -> (fusion l1 l2)::fusion_pair q
                | ll -> ll in (* si ll vide ou avec un seul élément *)
            kfusion (fusion_pair ll)
```

II Compression de Huffman

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).

Solution :

```
let rec read t l = match t, l with
    | F(c), _ -> c, l
    | N(g, d), e::q -> if e = 0 then read g q else read d q
```

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.

Solution :

```
let rec decode t = function
    if l = [] then []
    else let c, q = read t l in
        e::decode t q
```

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
}
```

3. Quel arbre obtient t-on avec les fréquences (20, ‘a‘), (15, ‘b‘), (7, ‘c‘), (14, ‘d‘), (44, ‘e‘)?
   ▶ `N(F(`e`), N(N(F(`c`), F(`d`)), N(F(`b`), F(`a`))))`

4. Écrire une fonction `to_huffman` construisant un arbre de Huffman à partir d'une FP vide et d'une liste de fréquences.
   ▶ En traduisant l'algorithme ci-dessus :

```
let to_huffman fp freq =
  let rec init = function
    | [] -> ()
    | (f, c)::q -> fp.add (f, F(c)); init q in
  init freq;
  let rec build () =
    let f1, a1 = fp.take_min () in
    if fp.is_empty () then a1
    else let f2, a2 = fp.take_min () in
         fp.add (f1 + f2, N(a1, a2));
         build ()
    in build ();;
```

On peut faire un peu plus simple en remarquant qu'à chaque itération de l'algorithme, la taille de la FP diminue de 1, donc qu'il faut un nombre d'itérations égal à `Array.length freq - 1` pour obtenir un seul arbre :

```
let to_huffman fp freq =
  let rec init = function
    | [] -> ()
    | (f, c)::q -> fp.add (f, F(c)); init q in
  init freq;
  for i = 0 to List.length freq - 2 do
    let (f1, a1), (f2, a2) = fp.take_min (), fp.take_min () in
    fp.add (f1 + f2, N(a1, a2))
  done;
  snd (fp.take_min ());;
```

5. Écrire une fonction `to_dico arb` stockant le code de chaque caractère dans un dictionnaire, à partir d'un arbre de Huffman `arb`. On utilisera le type de dictionnaire suivant :

```
type ('k, 'v) dico = {
    get : 'k -> 'v option;
    add : ('k*'v) -> unit;
}

val create : unit -> ('k * 'v) dico  (* créé un dico vide *)
```

▶ On peut utiliser un accumulateur (initialement vide) pour conserver le chemin (inversé) depuis la racine :

```
let rec to_dico arb di chemin = match arb with
  | F(e) -> di.add e (List.rev chemin)
  | N(g, d) -> to_dico g di (0::chemin); to_dico d di (1::chemin);;
```

6. En déduire une fonction pour coder un texte (sous forme d'une liste de lettres), à partir d'un dictionnaire (rempli par la fonction précédente).
▶

```
let rec code di = function
  | [] -> []
  | c::q -> (List.hd (di.get c)) @ code di q;;
```

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;;
```

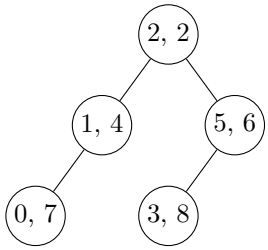
1. Écrire la fonction `swap : 'a array -> int -> int` utilisée par `shuffle`, telle que `swap t i j` échange `t.(i)` et `t.(j)`.
▶ Déjà faite en cours.

2. (Facultatif) Montrer que `shuffle t` applique une permutation choisie uniformément au hasard sur le tableau `t`.
▶ `shuffle t` applique un certain nombre de transpositions sur `t`, donc il effectue bien une permutation sur `t`.
Soit σ une permutation. On sait que σ est produit de transpositions donc on peut l'écrire $\sigma = (a_1 b_1)(a_2 b_2)...(a_p b_p)$ avec $a_k < b_k$ pour tout k et $b_1 < b_2 < ... < b_p$. Alors il existe exactement une possibilité pour que `shuffle t` applique σ sur `t` : que `Random.int (i+1)` renvoie a_1 pour `i` $= b_1$ (probabilité $\frac{1}{b_1}$), a_2 pour `i` $= b_2$, ... a_p si `i` $= b_p$ (probabilité $\frac{1}{b_p}$) et que `Random.int (i+1)` renvoie `i` (probabilité $\frac{1}{i}$) dans tous les autres cas.
La probabilité d'obtenir la permutation σ sur `t` est donc le produit de ces probabilités (ce sont des évènements indépendants), c'est à dire $\frac{1}{n!}$.

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 :

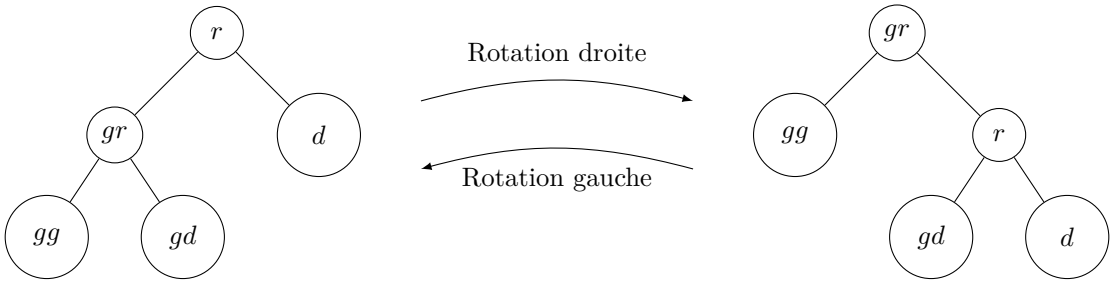
- les éléments doivent vérifier la propriété d'ABR.
- la priorité d'un sommet doit être inférieure à la priorité de ses éventuels fils (propriété de tas min sur les priorités).

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.
   ▶ Montrons P_n : « il existe un unique arbretas contenants n couples (élément, priorité) tous distincts » par récurrence sur n.
   P_0 est vraie : l'arbre vide convient et c'est le seul.
   Soit $n \in \mathbb{N}^*$ et supposons P_k vraie, $\forall k \leq n$. Considérons un ensemble C de $n+1$ couples (élément, priorité) tous distincts. Soit (e, p) le couple (élément, priorité) ayant la plus petite priorité, G les couples dont les éléments sont inférieurs à e et D les couples dont les éléments sont supérieurs à e. Les arbretas contenant les couples de C sont ceux s'écrivant $\mathrm{N}((e,p), g, d)$ avec g et d arbretas contenant G et D. D'après l'hypothèse de récurrence, il existe un unique tel g et un unique tel d. Donc il existe aussi un unique arbretas contenant les couples de C.

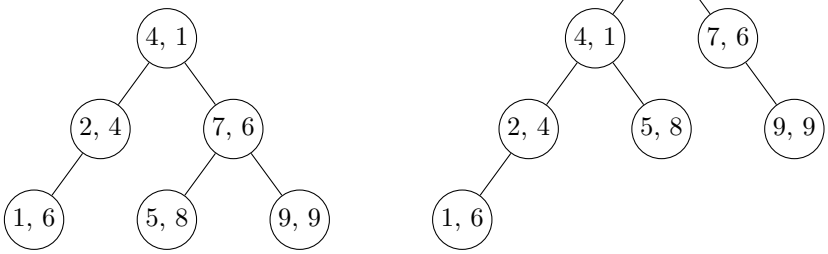
Nous allons utiliser des opérations de rotation sur un arbretas `N(r, N(gr, gg, gd), d)` :



5. Écrire une fonction `rotd` effectuant une rotation droite sur un arbre `N(r, N(gr, gg, gd), d)`.
   ▶ `let rotd = function N(r, N(gr, gg, gd), d) -> N(gr, gg, N(r, 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 :
   ▶ On obtient l'arbretas de droite.



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).

```
let prio = function
  | V -> max_int
  | N(r, g, d) -> snd r;;
```

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.

► `add` ajoute l'élément puis applique éventuellement une rotation sur le nouvel arbre :

```
let rec add a e = match a with
  | V -> N(e, V, V)
  | N(r, g, d) when e < r -> let g' = add g e in
                             if snd r < prio g' then N(r, g', d)
                             else rotd (N(r, g', d))
  | N(r, g, d) -> let d' = add d e in
                  if snd r < prio d' then N(r, g, d')
                  else rotg (N(r, g, d'));;
```

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.
   ► Il faut effectuer la rotation sur le fils de priorité minimum pour conserver la structure. On utilise le fait que `prio V` renvoie `max_int` pour s'assurer qu'une rotation est appliquée sur un arbre convenable (par ex. `g` doit être non vide pour appliquer `rotd N(r, g, d)`). Noter l'élégance du 4ème cas ci-dessous :

```
let rec del a e = match a with
  | N(r, g, d) when e < fst r -> N(r, del g e, d)
  | N(r, g, d) when fst r < e -> N(r, g, del d e)
  | N(r, V, V) -> V (* e = r *)
  | N(r, g, d) -> del (if prio g < prio d then rotd a else rotg a) e;;
```

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.
   ► On peut utiliser un accumulateur pour éviter `@`, comme dans le TD 1. On effectue un cons par élément de l'arbre, d'où une complexité linéaire.

```
let infixe =
  let rec aux acc = function
  | V -> acc
  | N(r, g, d) -> aux (r::aux acc d) g in
  aux [];;
```

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`.
   ► Le tableau du parcours infixe d'un ABR est croissant, donc représente un tas min (on a bien `t.(i) < t.(2*i+1)` et `t.(i) < t.(2*i+2)`).
   `let tas_of_abr a = Array.of_list (infixe a);;`

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.
   ► on peut trier `t` (en n `take` soit $O(n\log(n))$, puis construire l'ABR en $O(n)$ par dichotomie (la racine est le milieu du tableau, le sous-arbre gauche la partie gauche du tableau et le sous-arbre droit la partie droite du tableau) pour obtenir un arbre presque complet (cf exercice de TD), donc de hauteur $O(\log(n))$:

```
let abr_of_tas t =
        let n = Array.length t in
        let tab = Array.make n t.(0) in (* tab va être un tri de t *)
        for i = 0 to n - 1 do tab.(i) <- take t done;
        let rec aux i j = (* convertit tab.(i), ..., tab.(j-1) en ABR  *)
             if i >= j then V
             else let m = (i + j) / 2 in
                  N(tab.(m), aux i m, aux (m+1) j)
        in aux 0 n;;
```

Pour vérifier qu'on obtient bien un arbre binaire presque complet (i.e. que tous les niveaux sont complets sauf éventuellement le dernier), on peut montrer par induction structurelle que, si a est un arbre binaire dont tous les noeuds $\mathrm{N}(r, g, d)$ vérifient $|n(g) - n(d)| \leq 1$ (où $n(g)$ est le nombre de noeuds de g) alors a est presque complet.

4. Peut-on faire mieux que $\mathrm{O}(n \log(n))$ pour la question précédente?
   ▶ Si c'était le cas, on pourrait trier un tableau `t` de taille n en $\mathrm{o}(n \log(n))$ (ce qui est impossible) :

   i) transformer `t` en tas min, avec l'algorithme linéaire du cours, en $\mathrm{O}(n)$

   ii) convertir ce tas en ABR, en $\mathrm{o}(n \log(n))$

   iii) renvoyer le parcours infixe de cet ABR, en $\mathrm{O}(n)$