DS 1
Option informatique : corrigé

Les exercices sont indépendants, et ne sont pas forcément triés par difficulté croissante...
Toutes les complexités doivent être justifiées.

I Parcours préfixe

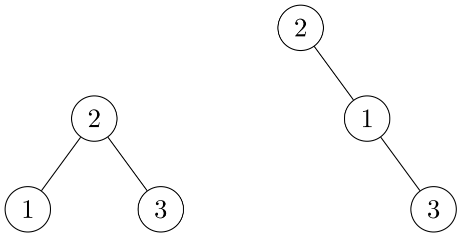
On définit un arbre binaire par : `type 'a arb = V | N of 'a * 'a arb * 'a arb;;`
On rappelle que le parcours préfixe d'un arbre binaire consiste à d'abord visiter sa racine, puis réaliser le parcours préfixe de son sous-arbre gauche et enfin le parcours préfixe de son sous-arbre droit.

1. Écrire une fonction `prefixe : 'a arb -> 'a list` renvoyant la liste des sommets d'un arbre parcouru dans l'ordre préfixe. Quelle est la complexité de votre fonction dans le pire cas ?
   ▶ Soit `a` un arbre binaire à n sommets. La fonction `prefixe a` ci-dessous effectue un appel récursif par sommet de `a` et chaque appel effectue un `@` dont la complexité est au plus n. Donc `prefixe a` est en $O(n^2)$ (on pourrait réduire la complexité à $O(n)$ en utilisant un accumulateur).

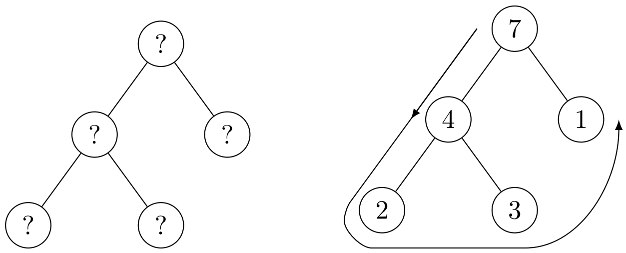
```
let rec prefixe a = match a with
  | V -> []
  | N(r, g, d) -> r::(prefixe g @ prefixe d);;
```

Soit `a` un arbre binaire étiqueté par des entiers tous différents et `p` sa liste de sommets dans l'ordre préfixe. On veut savoir si `p` détermine `a`, c'est à dire s'il existe un autre arbre avec le même parcours préfixe.

2. Sans hypothèse supplémentaire sur `a`, est-ce que `p` détermine `a` ?
   ▶ Non, les deux arbres suivants ont le même parcours préfixe `[2; 1; 3]` :



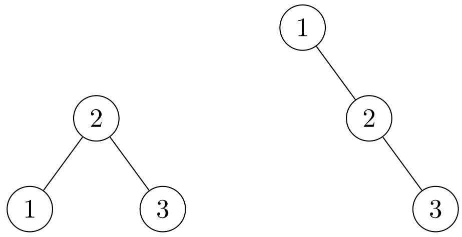
3. Supposons que `a` soit un tas max. Peut-il exister un autre tas max avec le même parcours préfixe `p` que `a` ?
   ▶ Soit n la taille de `p`. Comme `a` est presque complet et rempli « de gauche à droite », l'emplacement des sommets de `a` est déterminé. Alors les étiquettes des sommets sont aussi déterminés : la racine est `p.(0)`, le fils gauche est `p.(1)`...
   Par exemple, si `p = [7; 4; 2; 3; 1]`, le tas a 5 sommets est de la forme suivante (figure gauche) :



   Le parcours préfixe donne alors la seule façon de placer les étiquettes (figure droite ci-dessus).

4. Supposons que `a` soit un arbre binaire de recherche. Peut-il exister un autre arbre binaire de recherche avec le même parcours préfixe que `a` ?
   ▶ Soit $\mathcal{H}(n)$: « un ABR à n sommets est déterminé par son parcours infixe ». $\mathcal{H}(1)$ est trivialement vraie.
   Supposons $\mathcal{H}(k)$ pour $k \leq n$ et soit `p` le parcours préfixe d'un ABR `a = N(r, g, d)` à $n+1$ sommets. Alors nécessairement `r = p.(0)`. Soit `p1` (resp. `p2`) les éléments de `p` inférieurs (resp. supérieurs) à `r`. Comme `a` est un ABR, `g` contient les éléments de `p1`. `p1` est donc le parcours préfixe de `g`. Comme `g` a moins de sommets que `a`, par hypothèse de récurrence, `g` est déterminé par `p1`. De même pour `d`. Donc $\mathcal{H}(n)$ est vraie, ce qui termine la preuve.

5. Reprendre les 3 questions précédentes en remplaçant « préfixe » par « infixe ».
   ▶ les deux ABR suivants ont le même parcours infixe `[1; 2; 3]` :



   Par contre, le même raisonnement que 3. montre qu'un tas max est aussi déterminé par son parcours infixe.

II Extrait Centrale 2017

II.A – On veut implémenter une file d'attente à l'aide d'un vecteur circulaire. On définit pour cela un type particulier nommé `file` par

```
type 'a file={tab:'a vect; mutable deb: int; mutable fin: int; mutable vide: bool}
```

`deb` indique l'indice du premier élément dans la file et `fin` l'indice qui suit celui du dernier élément de la file, `vide` indiquant si la file est vide. Les éléments sont rangés depuis la case `deb` jusqu'à la case précédent `fin` en repartant à la case 0 quand on arrive au bout du vecteur (cf exemple). Ainsi, on peut très bien avoir l'indice `fin` plus petit que l'indice `deb`. Par exemple, la file figure 5 contient les éléments 4, 0, 1, 12 et 8, dans cet ordre, avec `fin=2` et `deb=9`.

		fin ↓							deb ↓		
12	8	7	2	5	3	1	16	3	4	0	1

Figure 5 Un exemple de file où `fin` < `deb`

On rappelle qu'un champ mutable peut voir sa valeur modifiée. Par exemple, la syntaxe `f.deb <- 0` affecte la valeur 0 au champ `deb` de la file `f`.

II.A.1) Écrire une fonction `ajoute` de signature `'a file -> 'a -> unit` telle que `ajoute f x` ajoute `x` à la fin de la file d'attente `f`. Si c'est impossible, la fonction devra renvoyer un message d'erreur, en utilisant l'instruction `failwith "File pleine"`.

II.A.2) Écrire une fonction `retire` de signature `'a file -> 'a` telle que `retire f` retire l'élément en tête de la file d'attente et le renvoie. Si c'est impossible, la fonction devra renvoyer un message d'erreur.

II.A.3) Quelle est la complexité de ces fonctions ?

▶ **II.A.1)**
```
let ajoute f e =
     if f.deb = f.fin && not f.vide then failwith "File pleine"
     else (f.tab.(f.fin) <- e; f.fin <- (f.fin + 1) mod (vect_length f.tab); f.vide <- false);;
```
II.A.2)
```
let retire f =
     if f.vide then failwith "File vide"
     else (let res = f.tab.(deb) in
           f.deb <- (f.deb - 1) mod (vect_length f.tab);
           f.vide <- f.deb = f.fin; res);;
```
II.A.3) Clairement O(1) : le nombre d'opérations réalisées ne dépend pas de la taille de `f`.

III Ensembles

On souhaite implémenter une structure d'ensemble.

Avec une liste

1. Écrire une fonction `appartient : 'a -> 'a list -> bool` déterminant si un élément appartient à une liste. Quelle est sa complexité dans le pire cas ?
   ```
   ▶let rec appartient e l = match l with
        | [] -> false
   ```

```
    | t::q -> t = e || appartient e q;;
```
Dans le pire cas, `appartient e l` parcourt toute la liste `l` donc sa complexité est linéaire en la taille de `l`.

2. Écrire une fonction `est_ens : 'a list -> bool` déterminant si une liste représente bien un ensemble (c'est à dire : ne contient pas de doublon).
   ▶
   ```
   let rec est_ens l = match l with
       | [] -> true
       | t::q -> not (appartient t q) && est_ens q;;
   ```
3. Écrire une fonction `inclus : 'a list -> 'a list -> bool` telle que `inclus l1 l2` déterminant si l'ensemble des éléments de `l1` est inclus dans ceux de `l2`. Quelle est sa complexité dans le pire cas ?
   ▶
   ```
   let rec inclus l1 l2 = match l1 with
       | [] -> true
       | e::q -> not (appartient e l2) && inclus q l2;;
   ```
   Si n_1 et n_2 sont les tailles de `l1` et `l2`, `inclus l1 l2` appelle n_1 fois `appartient e l2` qui est en $\mathrm{O}(n_2)$, d'où une complexité totale $\mathrm{O}(n_1 n_2)$.
4. Écrire une fonction `inter : 'a list -> 'a list -> 'a list` renvoyant l'intersection de deux listes, contenant les éléments appartenant aux deux listes. Quelle est sa complexité dans le pire cas ?
   ▶
   ```
   let rec inter l1 l2 = match l1 with
       | [] -> []
       | e::q when appartient e l2 -> e::inter q l2
       | e::q -> inter q l2;;
   ```
   Complexité similaire à `inclus`.
5. Écrire une fonction `union : 'a list -> 'a list -> 'a list` renvoyant l'union de deux listes, contenant les éléments appartenant à une des deux listes. Quelle est sa complexité dans le pire cas ?
   ▶
   ```
   let rec union l1 l2 = match l1 with
       | [] -> l2
       | e::q when appartient e l2 -> union q l2
       | e::q -> e::union q l2;;
   ```
   Complexité similaire à `inclus`.

Avec un arbre binaire de recherche

Dans cette partie, on étudie une implémentation d'ensemble par un arbre binaire de recherche, de type :

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

6. Quelle restriction impose l'utilisation d'un arbre binaire de recherche sur le type des éléments ?
   ▶ On doit pouvoir disposer d'une relation d'ordre sur les sommets.
7. Écrire une fonction `ajout : 'a -> 'a arb -> 'a arb` pour ajouter un élément à un arbre binaire de recherche. Quelle est sa complexité dans le pire cas ?
   ▶ Voir cours.
8. Écrire une fonction `appartient : 'a -> 'a arb -> bool` déterminant si un élément appartient à un arbre binaire de recherche. Quelle est sa complexité dans le pire cas ?
   ▶ Voir cours.
9. Écrire une fonction `est_ens : 'a arb -> bool` déterminant si un arbre binaire de recherche représente bien un ensemble (c'est à dire : ne contient pas de doublon). Quelle est sa complexité dans le pire cas ?
   ▶ Si `r` apparaît deux fois dans `N(r, g, d)`, alors il est dans `g` (par propriété d'ABR) :
   ```
   let rec est_ens a = match a with
       | V -> true
       | N(r, g, d) -> not (appartient r g) && est_ens g && est_ens d;;
   ```
   Il y a un appel à `appartient` par sommet dans l'arbre, donc la complexité est $\mathrm{O}(n^2)$, où n est le nombre de sommets.
10. Écrire une fonction `inter : 'a arb -> 'a arb -> 'a arb` renvoyant l'intersection de deux arbres binaires de recherche, contenant les éléments appartenant aux deux arbres binaires de recherche. Quelle est sa complexité dans le pire cas ?
    ▶ On peut implémenter `inter` en calculant les parcours infixes des ABR, puis prendre l'intersection de ces listes triées, pour avoir un algorithme en O(taille des arbres) :

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

```
let rec aux l1 l2 = (* intersection de listes triées *)
  match l1, l2 with
  | [], _ -> []
  | _, [] -> []
  | e1::q1, e2::q2 when e1 = e2 -> e1::inter q1 q2
  | e1::q1, e2::q2 when e1 < e2 -> inter q1 l2
  | e1::q1, e2::q2 -> inter l1 q2;;
```

`let inter a1 a2 = aux (infixe a1) (infixe a2);;`

11. Écrire une fonction `union : 'a arb -> 'a arb -> 'a arb` renvoyant l'union de deux arbres binaires de recherche, contenant les éléments appartenant à au moins un des deux arbres binaires de recherche. Quelle est sa complexité dans le pire cas ?
    ▶ Similaire à la question précédente.
12. Est-ce que l'implémentation avec un arbre binaire de recherche est plus efficace qu'avec une liste, dans le pire cas ? Comment pourrait-on améliorer ces complexités ?
    ▶ On pourrait améliorer les complexités en utilisant un ABR équilibré (AVL...).

IV Calcul de rang

Étant donné un tableau `t` de taille n, on souhaite sélectionner le $k$ème plus petit (que l'on appelle **élément de rang *k***). Nous allons voir plusieurs approches, en effectuant éventuellement un prétraitement.

Méthode simple

1. Comment effectuer un prétraitement en $O(n\log(n))$ sur `t`, pour ensuite être capable d'obtenir l'élément de rang k en $O(1)$? On demande d'expliquer les grandes lignes d'un tel algorithme, mais pas de l'écrire en détail.
   ▶ On pourrait trier le tableau puis renvoyer l'élément d'indice $k-1$. Pour trier en $O(n\log(n))$ on peut utiliser un tri par tas par exemple... (cf cours).

Avec un tas min

Dans cette partie, on utilise un tas min représenté par un tableau.

2. Quel est la meilleur complexité de prétraitement que vous connaissez pour transformer `t` en tas ? On demande d'expliquer les grandes lignes d'un tel algorithme, mais pas de l'écrire en détail.
   ▶ Cours.
3. Écrire une fonction `extraire : 'a vect -> int -> 'a` telle que si `tas` est un tableau représentant un tas min avec `n` éléments, `extraire tas n` supprime et renvoie le minimum du tas. On garantira une complexité $O(\log(n))$.
   ▶ Cours.
4. En déduire une fonction `rang : 'a vect -> int -> int -> 'a` telle que `rang tas n k` renvoie l'élément de rang `k` dans un tas min avec `n` éléments représenté par `tas`. Quelle est sa complexité ?
   ▶
   ```
   let rang t n k =
     for i = 0 to (k - 1) do extraire t n done;
     extraire t n;
   ```
5. Comment pourrait-on modifier la fonction précédente pour qu'un appel à la fonction `rang` ne change pas les éléments du tas ? On réécrira la fonction en garantissant une complexité $O(k\log(n))$.
   ▶ On peut mémoriser (dans une liste) les éléments supprimés puis les remettre dedans.

Avec un arbre binaire de recherche

6. Comment obtenir l'élément de rang 1 d'un arbre binaire de recherche ? En quelle complexité ?
   ▶ Il suffit de regarder tout à gauche, en $O(h)$.

Pour récupérer l'élément de rang k quelconque dans un arbre binaire de recherche, on ajoute une information à chaque sommet s : le nombre de sommets du sous-arbre enraciné en s.
On utilise donc le type : `type 'a arb_rang = V | N of 'a * 'a arb_rang * 'a arb_rang * int;;`

7. Écrire une fonction pour ajouter un élément dans un `arb_rang`. Complexité ?
   ▶ Comme dans le cours mais en augmentant la taille de 1 là où on rajoute l'élément.
8. Écrire une fonction pour supprimer un élément dans un `arb_rang`. Complexité ?
   ▶ Comme dans le cours mais en diminuant la taille de 1 là où on rajoute l'élément.
9. Écrire une fonction pour récupérer l'élément de rang k dans un `arb_rang` en temps linéaire en sa hauteur (et indépendant de k).
   ▶ Si le sous-arbre gauche contient $k-1$ éléments, on renvoie la racine. Sinon, on s'appelle récursivement sur l'un des sous-arbres. Chaque appel récursif augmente la profondeur du sommet visité : il y a donc $O(h)$ tels appels, chacun en $O(1)$.

```
let sz = function
  | V -> 0
  | N(_, _, _, n) -> n;;
let rec get_kth a k = match a with
  | N(r, g, d, _) when k = sz g + 1 -> r
  | N(r, g, d, _) when k < sz g + 1 -> get_kth g k
  | N(r, g, d, _) -> get_kth d (k - sz g - 1);;
```

Avec un algorithme de partition similaire au tri rapide

Cette méthode consiste à chercher l'élément de rang k en choisissant un pivot p (normalement aléatoirement, mais ici on prendra le premier élément possible pour simplifier) puis en partitionnant le reste du tableau en deux : les éléments inférieurs à p et ceux supérieurs. Enfin on cherche récursivement dans l'un des deux sous-tableaux.

10. Écrire une fonction `partition` telle que `partition t i j` modifie le tableau `t` de sorte que, entre les indices i et j, il contienne d'abord les éléments inférieurs au pivot, puis le pivot, puis les éléments supérieurs. `partition` doit être en temps $\mathrm{O}(j - i)$ et, si possible, en complexité mémoire $\mathrm{O}(1)$ (c'est à dire sans création de tableau intermédiaire). `partition` devra renvoyer le nouvel indice du pivot.
    ▶ `ipivot` est l'indice où il faut rajouter le prochain élément inférieur au pivot. On met le pivot à sa bonne place seulement à la fin de l'algorithme.

	ipivot		k	
< pivot		> pivot		non visité

```
let partition t i j =
  let pivot = t.(i) and ipivot = ref i in
  for k = i + 1 to j do
    if t.(k) < pivot
    then (t.(!ipivot) <- t.(k);
          incr ipivot;
          t.(k) <- t.(!ipivot))
  done;
  t.(!ipivot) <- pivot;
  !ipivot;;
```

11. En déduire un algorithme pour obtenir l'élément de rang k dans un tableau de taille n. Quelle est sa complexité dans le pire des cas ? Dans le meilleur ?
    ▶ On partitionne le tableau puis on regarde si le $k$ème est dans la partie gauche ou droite. Dans le pire cas, l'intervalle de recherche est diminué de 1 à chaque itération donc on effectue n appels à `partition`, d'où une complexité $\Theta(n^2)$. Dans le meilleur cas, on trouve l'élément de rang k dès la première partition, en $\Theta(n)$.

```
let rec get_kth t k i j =
  let ipivot = partition t i j in
  let left = ipivot - i + 1 in
  if left = k then t.(ipivot)
  else if left > k then get_kth t k i (ipivot - 1)
  else get_kth t (k - left) (ipivot + 1) j;;
```

On peut montrer, de la même façon que pour le tri rapide, que cet algorithme est en complexité moyenne linéaire en la taille n du tableau.

Bilan

12. Faire un tableau avec la complexité de prétraitement et de recherche de l'élément de rang k, pour chacune des méthodes ci-dessus.