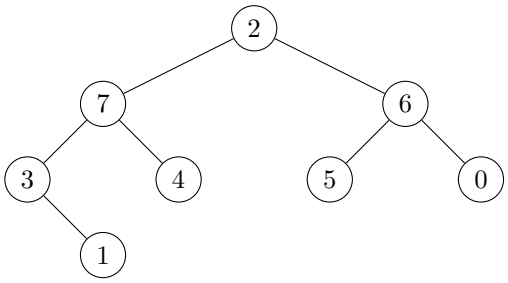
I Définitions

1. Donner l'ordre de chacun des parcours pour l'arbre suivant :



▶ Préfixe : 2, 7, 3, 1, 4, 6, 5, 0. Infixe : 3, 1, 7, 4, 2, 5, 6, 0. Suffixe : 1, 3, 4, 7, 5, 0, 6, 2.
En largeur : 2, 7, 6, 3, 4, 5, 0, 1.

2. Est-ce que le parcours postfixe est l'inverse (en lisant de droite à gauche) du parcours préfixe?
▶ Non: l'arbre ci-dessus est un contre-exemple.

3. Écrire une fonction `suffixe : 'a tree -> 'a list` renvoyant la liste des sommets d'un arbre dans l'ordre suffixe, si possible en complexité linéaire en le nombre de sommets.
▶ 1ère solution simple (mais quadratique):

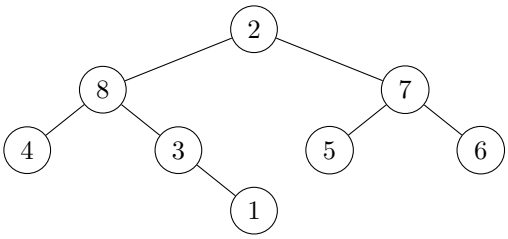
```
let rec suffixe = function
  | V -> []
  | N(r, g, d) -> suffixe g @ suffixe d @ [r];;
```

2ème solution avec accumulateur (linéaire): les sommets de g se retrouvent avant r, et ceux de d avant ceux de g.

```
let rec suffixe a acc = match a with
  | V -> acc
  | N(r, g, d) -> suffixe d (suffixe g (r::acc));;
```

II Reconstruction d'un arbre à partir du parcours préfixe et infixe

1. Quel est l'arbre binaire qui possède les tableaux de parcours préfixe `[2; 8; 4; 3; 1; 7; 5; 6]` et de parcours infixe `[4; 8; 3; 1; 2; 5; 7; 6]`?
▶ 2 est le premier élément du parcours préfixe donc doit être la racine. 8 est un fils de la racine comme c'est l'élément qui suit immédiatement 2 dans le parcours préfixe. De plus le parcours infixe nous dit que 8 est à gauche de 2 (car il est situé avant dans la liste). Donc 8 est le fils gauche de 2. Ainsi, de proche en proche, on construit l'arbre (déterminé de façon unique par cet algorithme de construction):



2. Écrire une fonction qui prend un tableau des éléments d'un arbre binaire parcouru dans l'ordre préfixe et un tableau des éléments du même arbre parcouru dans l'ordre infixe, et qui renvoie l'arbre correspondant. On supposera que les étiquettes de l'arbre sont des entiers tous différents.
▶

```
let rec reconstruire tp ti = (* tp: prefixe, ti: infixe *)
  let n = Array.length tp in
  match n with
  | 0 -> V
  | n -> let ind = ref 0 in
         (* ind va délimiter le sous-arbre gauche du sous-arbre droit dans ti *)
         for i = 0 to n - 1 do
           if ti.(i) = tp.(0) then ind := i
         done;
         let g = reconstruire (Array.sub tp 1 !ind) (Array.sub ti 0 !ind) in
         let d =  reconstruire (Array.sub tp (!ind + 1) (n - !ind - 1)) (Array.sub ti
(!ind + 1) (n- !ind - 1)) in
         N(tp.(0), g, d);;
```

3. Comment savoir si deux tableaux correspondent aux parcours préfixe et infixe d'un arbre binaire?
4. Est-il possible de reconstruire de façon unique un arbre binaire à partir de son parcours préfixe et postfixe?

III Parcours infixe d'un arbre binaire de recherche

1. Écrire une fonction `est_abr` permettant de déterminer si un arbre a est un ABR.
   On pourra renvoyer 3 informations (utiles pour les appels récursifs) : un booléen déterminant si a est un ABR, l'étiquette minimum de a et l'étiquette maximum de a.
   ▶

```
let est_abr a = est_croissant (infixe a);;
```

2. Quelle est la complexité de `est_abr`?

Dans la suite, on étudie une autre méthode de déterminer si un arbre est un ABR.

3. Notons $i(a)$ le parcours infixe d'un arbre a. Montrer qu'un arbre a est un ABR si et seulement si $i(a)$ est croissant.
   ▶ On montre par induction structurelle[1], pour tout arbre a, $\mathcal{P}(a)$: « a est un ABR $\iff$ $i(a)$ est croissant ».
   C'est vrai si a est l'arbre vide (un arbre vide est bien un ABR et une liste vide est bien croissante).
   Soit $a = N(r, g, d)$. Supposons $\mathcal{P}(g)$ et $\mathcal{P}(d)$.
   Montrons d'abord « a est un ABR $\implies$ $i(a)$ est croissant ». Si a est un ABR, g et d sont aussi des ABR (cette propriété de sous-structure est évidente d'après la définition). D'après $\mathcal{P}(g)$ et $\mathcal{P}(d)$, $i(g)$ et $i(d)$ sont croissants. Comme $i(a)$ est la concaténation de $i(g)$, r et $i(d)$, et que les éléments de $i(g)$ sont inférieurs à r, lui-même inférieur aux éléments de $i(d)$, on en déduit que $i(a)$ est bien croissant.
   Montrons maintenant « $i(a)$ est croissant $\implies$ a est un ABR ». Comme $i(a)$ est la concaténation de $i(g)$, r et $i(d)$, on en déduit que $i(g)$ et $i(d)$ sont croissants et que r est supérieur aux éléments de g et inférieur aux éléments de d. De plus, d'après $\mathcal{P}(g)$ et $\mathcal{P}(d)$, g et d sont des ABR. On en déduit que a est bien un ABR.
4. Écrire une fonction `croissant : int list -> bool` déterminant si une liste est triée par ordre croissant.
   ▶ On regarde si le premier élément est inférieur au 2ème et si la liste privée de son 1er élément est croissant. Il faut donc aussi distinguer la cas d'une liste à un seul élément. Attention aussi à s'appeler récursivement sur `e2::q` et non pas `q`.

```
let rec est_croissant l = match l with
  | [] -> true
  | [e] -> true
  | e1::e2::q -> e1 <= e2 && est_croissant (e2::q);;
```

5. En déduire une fonction pour déterminer si un arbre est un ABR. Quelle est sa complexité?
   ▶ On regarde si le parcours infixe de l'arbre est croissant, d'après 1. (on utilise ici un accumulateur pour la fonction infixe comme dans le TD précédent, afin d'avoir une complexité linéaire en le nombre de noeuds):

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

```
let est_abr a = est_croissant (infixe a);;
```

[1] On pourra aussi raisonner par récurrence sur le nombre de noeuds, par exemple

6. Autre application: écrire une fonction pour trier une liste, en utilisant un ABR. Quelle est sa complexité dans le pire des cas?
   ▶ On convertit la liste en ABR puis on renvoie son parcours infixe (qui est croissant d'après 1.). Comme `infixe` et `abr_of_list` sont en complexité linéaire, la fonction `tri` ci-dessous est aussi en complexité linéaire en la taille de la liste :

```
let tri_abr l =
    infixe (abr_of_list l)
```

IV Calcul de rang (order statistics)

Étant donné un tableau `t` de taille n, on souhaite sélectionner le $k$ème plus petit (que l'on appelle élément de rang k).

Méthode simple

1. Comment effectuer un prétraitement en $O(n \log(n))$ sur le tableau, pour ensuite être capable d'obtenir l'élément de rang k en $O(1)$?
   ▶ Utiliser un tri.

Avec un ABR

2. Comment obtenir l'élément de rang 1 d'un ABR? En quelle complexité?
   ▶ En regardant tout à gauche de l'arbre (complexité $O(h)$):

```
let rec abr_min = function
  | V -> failwith "arbre vide"
  | N(r, V, d) -> r
  | N(r, g, d) -> abr_min g;;
```

Pour récupérer l'élément de rang k quelconque dans un ABR, 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;;`

3. 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:

```
let rec add a e = match a with
  | V -> N(e, V, V, 1)
  | N(r, g, d, n) -> if e < r then N(r, add g e, d, n+1)
                     else N(r, g, add d e, n+1);;
```

4. 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);;
```

5. Écrire une fonction pour supprimer un élément dans un `arb_rang`. Complexité?
   ▶ On peut utiliser l'une des 2 méthodes vues en cours, par exemple par fusion :

```
let sz = function
  | V -> 0
  | N(_, _, _, n) -> n;;
let rec fusion g d = match d with
  | V -> g
  | N(rd, gd, dd, nd) -> N(rd, fusion g gd, dd, nd + sz g);;

let rec del a e = match a with
  | N(r, g, d, n) when e = r -> fusion g d
  | N(r, g, d, n) when e < r -> N(r, del g e, d, n-1)
  | N(r, g, d, n) -> N(r, g, del d e, n-1);;
```

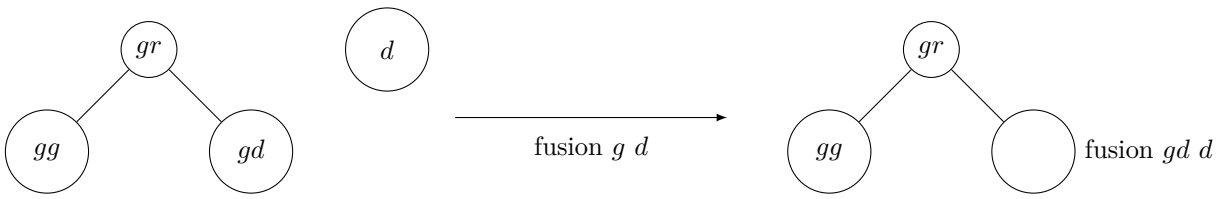
V Fusion d'ABR

On veut fusionner deux ABR de tailles n_1 et n_2, i.e obtenir un ABR constitué des éléments des deux ABR.

1. (Méthode naïve) Quelle est la méthode la plus simple qui vous vient à l'esprit pour fusionner deux ABR? L'implémenter.
   ▶ Ajouter (avec la fonction `add` du cours) un à un les éléments du 1er ABR au 2ème. Complexité: $O(n_1(n_1+n_2))$.

```
let rec fusion_naif a1 a2 = match a1 with
  | V -> a2
  | N(r, g, d) -> add r (fusion_naif g (fusion_naif d a2));;
```

2. (Cas simple) Écrire une fonction en $O(h)$ pour fusionner deux ABR g et d de hauteurs $\leq h$, en supposant que tous les éléments de g sont inférieurs à ceux de d. En déduire une fonction pour supprimer une valeur dans un ABR.
   ▶ On peut ajouter d tout à droite de g, ce qui donne bien un ABR (bien sûr, on peut aussi ajouter g tout à gauche de d). En décomposant $g = \texttt{N}(gr, gg, gd)$, cela revient à effectuer:




```
let rec fusion g d = match g with
  | V -> d
  | N(gr, gg, gd) -> N(gr, gg, fusion gd d);;

let rec del e = function
  | V -> V
  | N(r, g, d) when e = r -> fusion g d
  | N(r, g, d) when e < r -> N(r, del e g, d)
  | N(r, g, d) -> N(r, g, del e d);;
```

   Pour supprimer la racine d'un ABR, il suffit alors de fusionner ses deux fils.

3. Écrire une fonction renvoyant la liste infixe des sommets d'un arbre à n sommets.
   ▶ Si on ne se préoccupe pas de la complexité, on peut utiliser `@` (ce qui donne une complexité quadratique dans le cas d'un « peigne », où `g` a $n-1$ noeuds):

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

   Pour avoir une complexité linéaire au lieu de quadratique, on peut utiliser un accumulateur auquel on va rajouter les éléments dans l'ordre infixe :

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

   Il y a exactement un `::` par élément, et, comme ce sont les seules opérations réalisées, infixe est en $O(n)$.
   Rq: pour déterminer en temps linéaire si un arbre `a` est un ABR, on peut tester si `infixe a` est croissante.

4. Écrire une fonction prenant deux listes d'entiers triées et renvoyant leur fusion triée de taille n, en $O(n)$.
   ▶ Identique à la fonction utilisée pour le tri fusion:

```
let rec list_fusion l1 l2 = match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | e1::q1, e2::q2 when e1 < e2 -> e1::list_fusion q1 (e2::q2)
  | e1::q1, e2::q2 -> e2::list_fusion (e1::q1) q2;;
```

   Chaque appel récursif (qui effectue un nombre constant d'opérations) diminue de 1 le nombre total d'éléments des listes en arguments. Il y a donc au plus autant d'appels récursifs que la taille totale des listes: la fonction est linéaire en la taille totale des deux listes.

5. Écrire une fonction `array_to_abr` ayant un tableau **trié** `t` de taille n en argument et renvoyant un ABR dont les sommets sont étiquetés par les éléments de `t`, en $O(n)$.
   ▶ On peut construire l'ABR par dichotomie (ce qui donne un arbre presque complet): la racine est le milieu de `t`, et on s'appelle récursivement sur la partie gauche (resp. droite) pour construire le sous-arbre gauche (resp. droit).

```
let of_vect_trie t =
  let rec aux i j =
    if i >= j then V
    else let m = (i + j) / 2 in
         N(t.(m), aux i m, aux (m + 1) j)
  in aux 0 (vect_length t);;
```

   Il y a un appel récursif à `aux` par élément `t.(m)` du tableau, et, comme chaque appel fait un nombre constant d'opérations, la fonction est bien linéaire. On peut aussi résoudre $C(n) = 2C(\frac{n}{2}) + K$.
   Autre solution, en construisant un «peigne»:

```
let array_to_abr t =
  let rec aux i = (* renvoie un ABR contenant les éléments avant t.(i) *)
    if i = 0 then V
    else N(t.(i), aux (i-1), V) in
  aux (Array.length t);;
```

   Rq: le seul intérêt de passer par les tableaux est de permettre de faire une méthode par dichotomie (inefficace sur des listes car il faut accéder à l'élément du milieu). Avec la 2ème solution on pourrait donc rester avec des listes.

6. En déduire une fonction pour fusionner deux ABR de tailles n_1 et n_2 en complexité $O(n_1 + n_2)$. On pourra utiliser `Array.of_list : list -> array` pour convertir une liste en tableau, en temps linéaire.
   ▶ `let fusion a1 a2 = array_to_abr (Array.of_list (list_fusion (infixe a1) (infixe a2)));;`
   On effectue des opérations en temps linéaire les unes après les autres donc on somme les complexités.
   Remarque : je ne connais pas de façon simple de fusionner des ABR en temps linéaire sans passer par des listes!