TD structures de données 1 : corrigé partiel

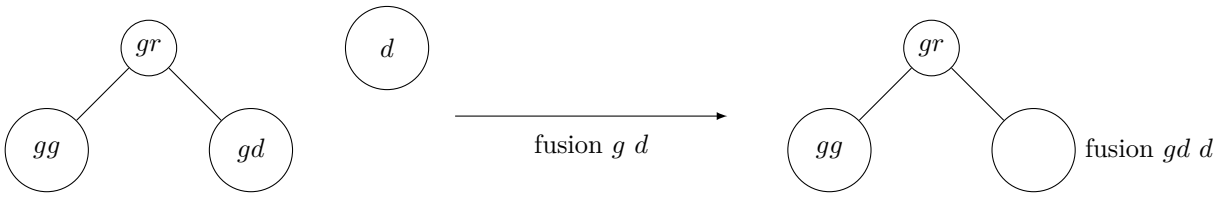
I Fusion d'ABR

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 différente de celle du cours 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 $g = N(gr, gg, gd)$, 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 en $O(n)$ la liste infixe des sommets d'un arbre à n sommets.
   ► Si l'énoncé ne demande pas que la complexité soit linéaire, 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));;
```

   Dans notre cas, on utilise un accumulateur auquel on va rajouter les éléments dans l'ordre infixe:

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

   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 array_to_abr t =
  let rec aux i j = (* renvoie un ABR contenant les éléments de t.(i) à t.(j) *)
    if i > j then V
    else let m = (i + j)/2 in
         N(t.(m), aux i (m-1), aux (m+1) j) in
  aux 0 (Array.length t - 1);;
```

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 $\Theta(n_1 + n_2)$. 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!

II Arbre AVL

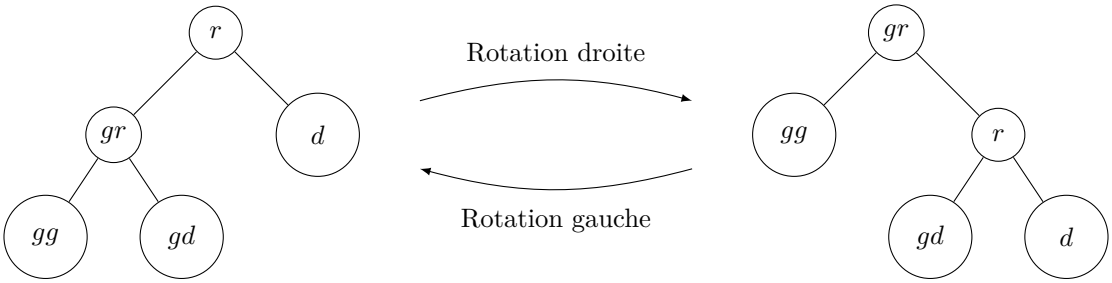
Un arbre AVL est un ABR tel que, pour chaque noeud, la différence de hauteur de ses sous-arbres gauche et droit soit au plus 1. Pour éviter de calculer plusieurs fois la même hauteur, on stocke, dans chaque noeud, la hauteur de l'arbre correspondant. On utilise donc le type suivant: `type 'a avl = V | N of 'a * 'a avl * 'a avl * int`.

1. Montrer qu'un arbre AVL à n noeuds est de hauteur $O(\log(n))$.
Indice: considérer le nombre de noeuds minimum u_h d'un arbre AVL de hauteur h.
► Soit $h \geq 1$ et `N(r, g, d)` un AVL de hauteur h ayant u_h noeuds. Alors `g` ou `d` est de hauteur $h-1$ et l'autre de hauteur $\geq h-2$, donc ont au moins u_{h-1} et u_{h-2} sommets, i.e $u_h \geq u_{h-1} + u_{h-2} + 1 \geq u_{h-1} + u_{h-2}$. Comme de plus $u_0 = 0 = f_0$, $u_h \geq f_h$, où f_h est la suite de Fibonacci définie par $f_n = f_{n-1} + f_{n-2}$ (on peut montrer $u_h \geq f_h$ par récurrence). Comme on sait que (cf équation récurrente d'ordre 2 du cours de maths) $f_n = \Theta(\phi^n)$ avec $\phi = \frac{1+\sqrt{5}}{2}$ (nombre d'or), $u_h \geq K\phi^h$ puis $h \leq \log_\phi u_h - K'$ pour des constantes K, K' et h assez grand. Donc la hauteur h d'un AVL à n sommets vérifie $h \leq \log_\phi u_h \leq \log_\phi n \approx 1.44 \log_2(n)$.

2. Écrire une fonction utilitaire `ht : 'a avl -> int` donnant la hauteur d'un AVL.
►
```
let ht a = match a with
      | V -> -1
      | N(r, g, d, h) -> h;;
```

3. Écrire une fonction utilitaire `node : 'a -> 'a avl -> 'a avl -> 'a avl` construisant un AVL à partir d'une racine et de ses deux sous-arbres.
► `let node r g d = N(r, g, d, 1 + max (ht g) (ht d));;`

Lors de l'ajout d'un élément (en tant que feuille) la condition d'AVL peut être violée et doit être rétablie. Pour cela, on introduit l'opération de *rotation droite* (et son symétrique *rotation gauche*) autour d'un noeud:



4. Est-ce qu'une rotation préserve la propriété d'ABR?
   ▶ Oui, on peut le vérifier sommet par sommet.

5. Écrire une fonction `rotd` réalisant une rotation droite sur un arbre supposé de forme convenable.
   ▶ Voici deux possibilités équivalentes, où on suppose que l'arbre donnée est de la même forme que ci-dessus à gauche:
   `let rotd (N(r, N(gr, gg, gd, _), d, _)) = node gr gg (node r gd d);;`
   `let rotd = function N(r, N(gr, gg, gd, _), d, _) -> node gr gg (node r gd d);;`

On suppose que, après l'ajout d'un élément, `g` et `d` sont des AVL, mais que `N(r, g, d)` n'est pas un AVL. Pour les deux questions suivantes, on suppose que `ht g > ht d + 1`, l'autre cas étant symétrique. On décompose `g` en `N(gr, gg, gd)`.

6. Si `ht gg > ht gd`, montrer qu'une rotation suffit pour transformer `N(r, g, d)` en AVL.
   ▶ Comme `ht gg > ht gd`, `ht g = ht gg + 1`. Comme l'ajout d'un élément augmente la hauteur d'au plus 1, `ht g = ht d + 2` et `ht gg = ht gd + 2`. On en déduit `ht gg = ht g - 1`, `ht gd = ht g - 3`, `ht d = ht d - 2`. D'où `ht (N(r, g, d)) = ht g - 2` et la condition d'AVL est bien respectée après rotation.

7. Sinon, `ht gg < ht gd`. Montrer comment se ramener au cas précédent en une rotation.
   ▶ On fait une rotation gauche sur g.

8. En déduire une fonction `balance` prenant `r`, `g`, `d` en arguments et renvoyant l'AVL correspondant.
   ▶ On remarque que la complexité de `balance` est O(1):

```
let balance r g d =
  if ht g > 1 + ht d then match g with
    | N(_, gg, gd, _) when ht gg > ht gd -> rotd (node r g d)
    | _ -> rotd (node r (rotg g) d) (* rotation gauche-droite *)
  else if ht d > 1 + ht g then match d with (* cas symetrique *)
    | N(dr, dg, dd, _) when ht dd > ht dg -> rotg (node r g d)
    | _ -> rotg (node r g (rotd d))
  else node r g d;;
```

9. En déduire une fonction `add` ajoutant un élément dans un AVL en conservant la structure d'AVL.
   ▶ On ajoute l'élément en conservant la propriété d'ABR puis on appelle `balance` pour obtenir à nouveau un AVL. Complexité: $O(h) = O(\log(n))$ d'après la question 1.

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

10. Écrire aussi des fonctions `del` et `mem` pour supprimer un élément et savoir si un élément appartient à un AVL. On supposera qu'il n'y a pas de doublon dans l'AVL. Complexité de ces fonctions?
    ▶ La méthode de suppression du cours est utilisée. Contrairement à la méthode de I.2, elle n'introduit qu'un déséquilibre d'au plus 1 donc on peut ensuite appliquer `balance` pour rééquilibrer l'arbre en AVL.
    `mem` est le même que pour les ABR. Complexités O($\log(n)$).

```
let rec del_max = function (* renvoie (max, arbre obtenu en supprimant max) *)
  | N(r, g, V, _) -> r, g
  | N(r, g, d, _) -> let m, d' = del_max d in
                     m, balance r g d';;

let del a e = match a with
  | V -> V (* l'élément à supprimer n'a pas été trouvé *)
  | N(r, g, d, _) when e < r -> balance r (del g e) d
  | N(r, g, d, _) when e > r -> balance r g (del d e)
  | N(r, g, d, _) -> let m, g' = del_max g in
                     balance m g' d;;
```

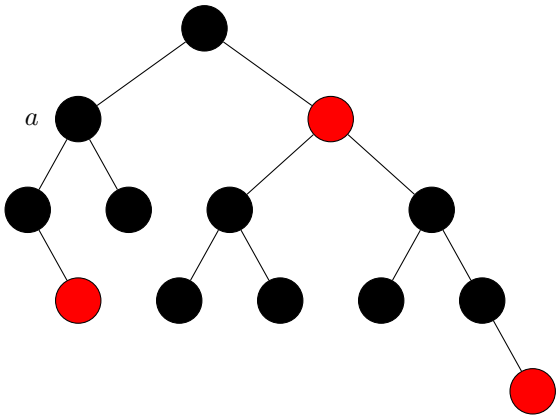
III Arbre rouge-noir

Un arbre rouge-noir (ARN) a est un ABR dont les sommets sont coloriés en rouge ou noir, et vérifiant les conditions suivantes:

(i) si un sommet est rouge, ses éventuels fils sont noirs.

(ii) le nombre de sommets noirs le long de n'importe quel chemin de la racine **à un emplacement vide** (`V`) est le même, que l'on appelle *hauteur noire* et note $h_b(a)$.

1. Colorier l'arbre ci-dessous (les labels ont été omis) pour qu'il devienne un ARN. Est-ce un AVL?
   ▶ Non: la condition d'AVL n'est pas vérifiée pour a.



2. Trouver un arbre binaire qui ne peut pas être colorié pour devenir un ARN.

3. Soit a ARN. On note $n(a)$ et $h(a)$ le nombre de sommets et la hauteur de a. Montrer que $h(a) \leq 2h_b(a)$ et $n(a) \geq 2^{h_b(a)} - 1$. En déduire que $h(a) = \mathrm{O}(\log(n(a)))$.

   ▶ Soit un chemin $\mathcal{C}$ de longueur $h(a)$ de la racine à une feuille. Alors $\mathcal{C}$ a au moins $\dfrac{h(a)}{2}$ sommets noirs donc $h_b(a) \geq \dfrac{h(a)}{2}$.

   Montrons $n(a) \geq 2^{h_b(a)} - 1$ par induction structurelle sur a:

   (a) C'est vrai pour un arbre a à 1 sommet car, dans ce cas, $h_b(a) \leq 1$ donc $2^{h_b(a)} - 1 \leq 2 - 1 = 1 = n(a)$.

   (b) Soit $a = N(r, g, d)$ un ARN. Comme g est aussi un ARN, on peut lui appliquer l'hypothèse d'induction: $n(g) \geq 2^{h_b(g)} - 1$ En raisonnant de la même façon avec d, on en déduit $n(a) = n(g) + n(d) + 1 \geq (2^{h_b(a)-1} - 1) + (2^{h_b(a)-1} - 1) + 1 = 2^{h_b(a)} - 1$.

   De $n(a) \geq 2^{h_b(a)} - 1$ on déduit que $h_b(a) \leq \log_2(n+1)$ puis $h(a) \leq 2h_b(a) \leq 2\log_2(n+1)$, ce qui montre que $\boxed{h(a) = O(\log(n(a)))}$.

Un arbre rouge-noir est représenté par le type (`R`: rouge, `N`: noir):

```
type 'a arn = V | R of 'a * 'a arn * 'a arn | N of 'a * 'a arn * 'a arn;;
```

On dira qu'un arbre est rouge (resp. noir) si sa racine est rouge (resp. noire).

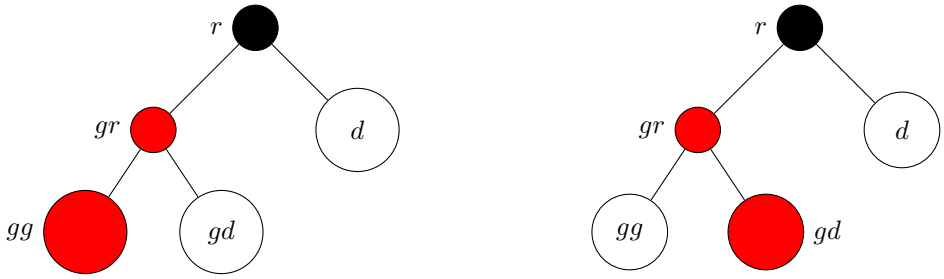
4. Écrire une fonction `noircir : 'a arn -> 'a arn` changeant la couleur d'un arbre pour qu'il devienne noir.
   ▶

```
let noircir a = match a with
  | R(r, g, d) -> N(r, g, d)
  | _ -> a;;
```

5. Écrire une fonction `rouge : 'a arn -> bool` déterminant si un arbre est rouge.
   ▶

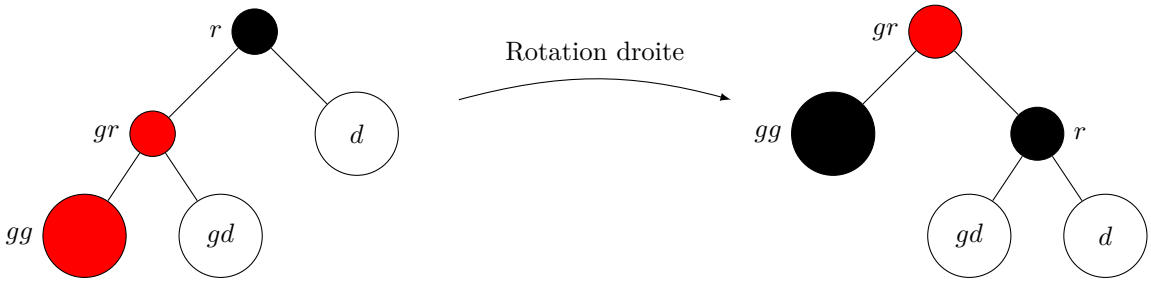
```
let rouge = function
  | R(_, _, _) -> true
  | _ -> false;;
```

On souhaite ajouter un élément dans un ARN, en tant que feuille rouge. On remarque alors que (i) peut être violé, mais pas (ii). Il existe donc un sommet rouge de père rouge, qui lui-même a un père noir. Considérons 2 situations possibles, où on suppose vérifiées les conditions d'ARN pour les sous-arbres gg, gd et d (on colorie un arbre de la même couleur que sa racine):



6. Que peut-on dire de $h_b(gg)$, $h_b(gd)$ et $h_b(d)$ dans ces deux situations?
   ► Ils sont égaux, d'après la propriété (ii) d'ARN.

7. Montrer que l'on peut rétablir la condition d'ARN en 1 rotation (même opération que dans l'exercice précédent) dans le cas de gauche, 2 dans le cas de droite (en changeant éventuellement la couleur de certains noeuds).
   ► Pour la situation de gauche (d et gd garde la même couleur), on effectue la transformation:



   On peut vérifier que les conditions (i) et (ii) sont alors vérifiées.
   Pour le cas de droite, on peut d'abord effectuer une rotation gauche sur $N(r, gg, gd)$ pour retomber dans le 1er cas.

8. Quelles sont les autres situations possibles? En déduire une fonction `balance : 'a arn -> 'a arn` effectuant la bonne transformation.
   ► Il y a les cas symétriques où $d = N(dr, dg, dd)$ où dr est rouge et soit dg soit dd est rouge. Ici j'utilise une fonction `balance` récursive juste pour s'appeller récursivement lorsqu'il faut effectuer deux rotations:

```
let rec balance (N(r, g, d)) = match g, d with
  | R(gr, gg, gd), d when rouge gg -> R(gr, noircir gg, N(r, gd, d))
  | R(gr, gg, gd), d when rouge gd -> balance (N(r, rotg g, d))
  | g, R(dr, dg, dd) when rouge dd -> R(dr, N(r, g, dg), noircir dd)
  | g, R(dr, dg, dd) when rouge dg -> balance (N(r, g, rotd d))
  | _, _ -> N(r, g, d);;
```

   Remarque: en mettant `N(r, g, d)` en argument on n'a pas besoin de match supplémentaire mais la fonction ne marche pas pour l'arbre vide.

9. En déduire une fonction `add` ajoutant un élément dans un ARN en conservant la structure d'ARN. Complexité?
   ► On ajoute le sommet en tant que feuille au bon endroit de façon à conserver la propriété d'ABR puis on applique `balance` pour rétablir les propriétés d'ARN:

```
let rec add a e = match a with
  | V -> R(e, V, V)
  | N(r, g, d) when e < r -> balance (N(r, add g e, d))
  | R(r, g, d) when e < r -> R(r, add g e, d)
  | N(r, g, d) -> balance (N(r, g, add d e))
  | R(r, g, d) -> R(r, g, add d e);;
```

   `balance` est en O(1) (au plus 1 appel récursif et un nombre constant d'opérations). Chaque appel récursif de `add` s'effectue sur un sommet de profondeur plus grande. Comme la profondeur d'un sommet est limitée par la hauteur de l'arbre, le nombre d'appels récursifs de `add a e` est au plus $O(h)$ où h est la hauteur de `a`. Chacun de ces appels effectue par ailleurs un nombre constant d'opérations, donc `add a e` est en $O(h)$. D'après 3., ls effectue par ailleurs un nombre constant d'opérations, donc est aussi en $O(\log(n))$, où n est le nombre de sommets de `a`.

IV Dictionnaire et ensemble

On rappelle les types abstraits de dictionnaire et ensemble vus en cours:

```
type ('key, 'value) dico =
  { add : 'key -> 'value -> unit;
    get : 'key -> 'value list;
    del : 'key -> unit };;
```

```
type 'a set =
  { add : 'a -> unit;
    del : 'a -> unit;
    has : 'a -> bool };;
```

1. Expliquer comment implémenter un dictionnaire en utilisant un ABR. Écrire la fonction `get` correspondante.
   ▶ On stocke les couples (clé, valeur) dans les sommets et on compare les sommets en utilisant les clés. On peut utiliser la fonction suivante pour implémenter `get` (ici avec un AVL):

```
let rec get a k = match a with
  | V -> []
  | N(r, g, d, _) when k < fst r -> get g k
  | N(r, g, d, _) when k > fst r -> get d k
  | N(r, g, d, _) -> [snd r];;
```

   `add` et `del` ressemblent beaucoup aux fonctions du cours.

2. Expliquer comment connaître l'élément le plus fréquent dans une liste de taille n, en complexité $O(n\log(n))$. Comment obtenir une complexité moyenne linéaire?
   ▶ On peut utiliser un dictionnaire implémenté par AVL pour stocker les éléments avec leurs nombres d'apparitions. On effectue au plus 3 get, 1 del et 1 add par élément, chacune en $O(\log(n))$ car la hauteur d'un AVL est $O(\log(n))$. D'où la complexité $O(n\log(n))$. Une implémentation par table de hachage donne une complexité moyenne linéaire.
   Voici à quoi pourrait ressembler une implémentation de cette méthode, où `of_avl V` donne un `dico` implémenté par AVL:

```
let freq l = (* renvoie l'élément le plus frequent dans l *)
  let d = of_avl V in
  let rec add_d = function
    | [] -> ()
    | e::q -> let g = d.get e in
              if g <> [] then begin d.del e; d.add (e, hd g + 1) end
              else d.add (e, 1)
  in add_d l; (* ajoute les elements et leur frequence dans le dico *)
  let rec maxi = function
    | [e] -> d.get e, e
    | e::q -> let m2, e2 = maxi q and m1 = d.get e in
              if m1 > m2 then m1, e else m2, e2
  in snd (maxi l);; (* on renvoie la clé de valeur maximum *)
```

3. Écrire un algorithme en complexité $O(n\log(n))$ pour savoir si une liste de n entiers relatifs contient une séquence d'éléments consécutifs de somme nulle.
   ▶ Il suffit de stocker, dans un `set`, toutes les sommes des k premiers éléments de la liste. Si on trouve deux sommes égales $\sum_{i=0}^{k} a_i = \sum_{i=0}^{\ell} a_i$ avec $\ell > k$ alors $\sum_{i=k+1}^{\ell} a_i = 0$ et il faut renvoyer `true`. Sinon on renvoie `false`.
   Voici une implémentation possible, avec `set_create : unit -> 'a set` une fonction de création de `set` vide:

```
let somme_nulle l =
  let s = set_create () in (* contient les sommes partielles *)
  let sum = ref 0 in (* somme partielle en cours de calcul *)
  s.add 0; (* pour que l'on renvoie true si on trouve une somme partielle nulle *)
  let rec aux = function
    | [] -> false
    | e::q -> sum := !sum + e;
              if s.has !sum then true (* on a trouvé 2x la même somme *)
              else (s.add !sum;
                    aux q) in
  aux l;;
```

   En utilisant un `set` implémenté par un AVL, les opérations d'ajouts (`add`) et de test d'appartenance (`has`) sont en $O(\log(n))$, car le `set` contient au plus n éléments. En effectuant au plus n `add` et n `has`, on obtient un algorithme en $O(n\log(n))$.