I 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 $\mathrm{O}(\log(n))$.

> **Solution** : Si a est un AVL à n noeuds et de hauteur h, on veut montrer $n \geq A^h$ ($\Longleftrightarrow h \leq \log_A(n)$), où M est une constante que l'on va déterminer ultérieurement (de façon à ce que la récurrence marche).
> Si $n \in \mathbb{N}^*$, on pose H_n : « si a est un AVL à n noeuds et de hauteur h alors $n \geq M^h$ ».
> - H_1 est vraie car $0 \leq M^0$.
> - Supposons H_k

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 $\phi = \dfrac{1+\sqrt{5}}{2}$ (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 $h \leq \log_\phi u_h \leq \log_\phi n \approx 1.44 \log_2(n)$.

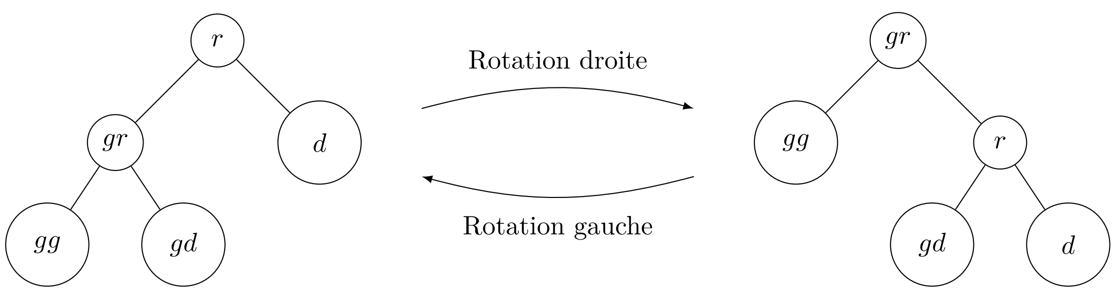
2. Écrire une fonction utilitaire `ht : 'a avl -> int` donnant la hauteur d'un AVL.

> **Solution** :
```
let rec ht = function
    | 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.

> **Solution** : `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?

> **Solution** : Oui, on peut le vérifier sommet par sommet.

5. Écrire une fonction `rotd` réalisant une rotation droite sur un arbre supposé de forme convenable.
On suppose dans la suite avoir aussi une fonction `rotg` réalisant une rotation gauche (opération inverse).

> **Solution** : 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.

> **Solution :** 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.

> **Solution :** On fait une rotation gauche sur g.

8. En déduire une fonction `balance` prenant `r`, `g`, `d` en arguments et renvoyant l'AVL correspondant.

> **Solution :**

```
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.

> **Solution :**

```
let rec add e = function 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 `has` 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?

> **Solution :** `has` est exactement comme pour les ABR classiques (pas besoin de rééquilibrer puisqu'on ne modifie par l'AVL). Pour `del`, on utilise la méthode du cours avec `del_max`. Comme on modifie la hauteur d'au plus 1 en supprimant un noeud, on peut utiliser ensuite `balance` pour rééquilibrer. Les deux fonctions sont en complexité O($logn$).

```
let rec has e = function
    | V -> false
    | N(r, g, d, h) -> r = e || has e (if e < r then g else d);;

let rec del_max = function
    | V -> min_int, V
    | N(r, g, V, _) -> r, g
    | N(r, g, d, h) -> let m, d' = del_max d in
                       m, balance r g d';;
let rec del e = function
    | V -> V
    | N(r, g, d, h) -> if e < r then balance r (del g e) d
                       else if e > r then balance r g (del d e)
                       else let g' = del_max g in
                            balance r g' d;;
```

II Implémentation de dictionnaire avec arbre

Écrire des fonctions `dict_get : 'a -> ('a * 'b) avl -> 'b option`,
`dict_add : 'a * 'b -> ('a * 'b) avl -> ('a * 'b) avl` `dico_del`, qui permettent d'implémenter un dictionnaire à l'aide

d'un AVL, en mettant un couple (clé, valeur) sur chaque noeud (`'a` étant le type des clés et `'b` celui des valeurs). Complexités ?
Remarque : On pourrait faire de même avec un ARN ou n'importe quel arbre binaire de recherche.

Solution :

```
let rec dict_get k = function
    | V -> None
    | N (r, _, _, _) when k = fst r -> Some (snd r)
    | N (r, g, _, _) when k < fst r -> dict_get k g
    | N (r, _, d, _) when k > fst r -> dict_get k d

let rec dict_add c = function
    | V -> N (c, V, V, 0)
    | N (r, g, d, _) -> if fst c < fst r then node r (dict_add c g) d
                        else node r g (dict_add c d)
```

III Problème de géométrie

On considère un ensemble E de n points dans le plan $\mathbb{R}^2$ (représenté par un tableau de couples, par exemple). Donner un algorithme pour trouver le nombre de rectangles que l'on peut former en choisissant 4 points de E. On pourra d'abord donner une solution simple en $O(n^4)$ puis une solution plus efficace, en utilisant un dictionnaire (quelle complexité obtient-on alors avec un dictionnaire implémenté par table de hachage? par arbre binaire de recherche équilibré ?).

Solution :

- En $O(n^4)$: Énumérer tous les quadruplets de points et vérifier si ils forment un rectangle.

```
let dist (x1, y1) (x2, y2) = (x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2);;

let is_rect p q r s =
    (* vérifie que p, q, r, s forme un rectangle dans cet ordre *)
    (dist p q) = (dist r s) && (dist q r) = (dist s p);;

let rectangles ens =
    let n = Array.length ens in
    let res = ref 0 in
    for i = 0 to n - 1 do
        for j = i + 1 to n - 1 do
            for k = j + 1 to n - 1 do
                for l = k + 1 to n - 1 do
                    if is_rect ens.(i) ens.(j) ens.(k) ens.(l)
                    then incr res
                done
            done
        done
    done;
    !res;;
```

- En $O(n^2)$: Remarquer que 4 points forment un rectangle ssi les deux diagonales sont de même longueur et ont même milieu. On peut donc utiliser un dictionnaire, pour stocker le nombre de fois que l'on a obtenu un couple (longueur, milieu) par les paires de points :

```ocaml
let middle (x1, y1) (x2, y2) = ((x1 +. x2)/.2., (y1 +. y2)/.2.);;

let rectangles ens =
    let d = ref V in
    let n = Array.length ens in
    let res = ref 0 in
    for i = 0 to n - 1 do
        for j = i + 1 to n - 1 do
            let l = dist ens.(i) ens.(j) in
            let m = middle ens.(i) ens.(j) in
            d := match dico_get (l, m) !d with
                | None -> dico_add ((l, m), 1) !d
                | Some n -> dico_del (l, m) !d |> dico_add ((l, m), n+1);
            res := res + dico_get (l, m) !d
        done
    done;
    !res;;
```