TP 2, Arbres AVL, et tas presque équilibrés : Corrigé

1 Arbres AVL : implémentation

1.1 Vérification des propriétés d'ABR

Question 1.

```
let rec enum_infixe a=match a with
  | Vide -> []
  | N(_,g,x,d) -> (enum_infixe g)@[x]@(enum_infixe d)
;;
```

Question 2.

```
let rec est_croissante_strict q=match q with
  | [_] | [] -> true
  | x::y::p -> x<y && est_croissante_strict (y::p)
;;
```

Question 3.

```
let check_abr a=est_croissante_strict (enum_infixe a) ;;
```

1.2 Calcul de hauteurs dans des quasi-AVL

Question 4.

```
let ha a=match a with
  | Vide -> -1
  | N(h,_,_,_) -> h
;;
```

Question 5.

```
let ha_bis a=match a with
  | Vide -> -1
  | N(_,g,_,d) -> 1+ max (ha g) (ha d)
;;

let recalc_ha a=match a with
  | Vide -> Vide
  | N(_,g,x,d) -> N(ha_bis a, g,x,d)
;;
```

Question 6. Un ABR vérifie la propriété d'AVL s'il est vide ou si :

- ses sous-arbres gauche et droit sont des AVL ;
- la différence de hauteurs entre les deux et d'au plus un ;
- le champ hauteur de l'arbre correspond bien à sa hauteur.

```
let rec check_avl a=match a with
  | Vide -> true
  | N(h,g,_,d) -> check_avl g && check_avl d && abs (ha g - ha d) <= 1 && h = 1+max (ha g) (ha d)
;;
```

1.3 Rotations, Equilibrage, Insertion et Suppression

Question 7. Il suffit d'appeler `recalc_ha` sur les arbres construits avec un champ hauteur arbitraire.

```
let rotation_d a=match a with
  | N(_, N(_,alpha,x,beta),y,gamma) -> recalc_ha (N(0,alpha,x, recalc_ha (N(0,beta,y,gamma))))
  | _ -> failwith "rd impossible"
;;
```

Question 8.

```
let rotation_g a=match a with
  | N(_,alpha,x,N(_,beta,y,gamma)) -> recalc_ha (N(0, (recalc_ha (N(0,alpha,x,beta))),y,gamma))
  | _ -> failwith "rg impossible"
;;
```

Question 9. *Équilibrage d'arbres*. De même puisque les fonctions de rotation recalculent les hauteurs sans regarder ce champ, on les applique sur un arbre où le champ hauteur est arbitraire.

```
let equilibrer a=match a with
  | N(_,g,x,d) when ha g = ha d + 2 -> begin match g with
    | N(_,alpha,y,beta) when ha alpha >= ha beta -> rotation_d a
    | _ -> rotation_d (N(0,rotation_g g,x,d))
  end
  | N(_,g,x,d) when ha d=ha g + 2 -> begin match d with
    | N(_,alpha,y,beta) when ha beta >= ha alpha -> rotation_g a
    | _ -> rotation_g (N(0,g,x,rotation_d d))
  end
  | _ -> recalc_ha a
;;
```

1.4 Insertion et suppression dans un arbre AVL

Question 10.

```
let rec max_abr a=match a with
  | Vide -> failwith "vide"
  | N(_,_,x,Vide) -> x
  | N(_,_,_,d) -> max_abr d
;;
```

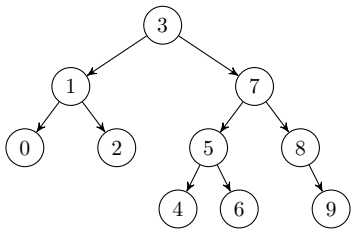
Question 11. On convient que si on veut insérer un nœud déja présent ou supprimer un nœud non présent, on renvoie l'arbre à l'identique.

```
let rec inserer a x=match a with
  | Vide -> N(0,Vide,x,Vide)
  | N(h,g,y,d) when y=x -> a
  | N(h,g,y,d) when y<x -> equilibrer (N(h,g,y,inserer d x))
  | N(h,g,y,d) -> equilibrer (N(h,inserer g x,y,d))
;;

let rec supprimer a x=match a with
  | Vide -> Vide
  | N(_,g,y,d) when y>x -> equilibrer (N(0,supprimer g x, y, d))
  | N(_,g,y,d) when y<x -> equilibrer (N(0,g, y, supprimer d x))
  | N(_,Vide,y,d) -> d
  | N(_,g,y,d) -> let z=max_abr g in equilibrer (N(0,supprimer g z, z, d))
;;
```

1.5 Test

Question 12. Chez moi, ça marche ! Pour l'exemple, voici l'arbre a 10 nœuds produit par insertions successives des éléments de $[\![0, 9]\!]$:



Dans un arbre ABR standard, on aurait obtenu un « peigne » (une unique branche droite).

2 Tas presque équilibrés

Question 13. Montrons la propriété par récurrence sur le nombre de nœuds.

- Pour $n = 1$, l'arbre est réduit à une racine, donc de hauteur $0 = \lfloor \log_2(1) \rfloor$;
- Supposons la propriété vérifiée pour tout entier jusqu'à $n \geq 2$ exclu. Considérons un APE à n nœuds. Alors sa hauteur est égale au maximum des hauteurs de ses sous-arbres gauche et droit plus 1. Par hypothèse de récurrence, celle-ci vaut $1 + \left\lfloor \log_2 \left(\left\lceil \frac{n-1}{2} \right\rceil \right) \right\rfloor = \left\lfloor \log_2 \left(2 \left\lceil \frac{n-1}{2} \right\rceil \right) \right\rfloor$.
  - Si n est impair, $\left\lceil \frac{n-1}{2} \right\rceil = \frac{n-1}{2}$, donc cette hauteur vaut $\lfloor \log_2(n-1) \rfloor = \lfloor \log_2(n) \rfloor$.
  - Si n est pair, $\left\lceil \frac{n-1}{2} \right\rceil = \frac{n}{2}$, et la hauteur est précisément $\lfloor \log_2(n) \rfloor$.
- Par principe de récurrence, la propriété est démontrée.

Question 14. Pour tester si un arbre est un TPE, on écrit une fonction auxiliaire prenant en entrée un arbre est renvoyant un couple (b, n) où b est un booléen indiquant si l'arbre est un TPE et n son nombre de nœuds. Ainsi, un arbre a est un TPE s'il est vide, ou si :

- ses deux sous-arbres gauche et droit sont des TPE ;
- la condition sur le nombre de nœuds des sous-arbres gauche et droit est vérifiée ;
- la condition de tas est vérifiée, c'est-à-dire que son sous-arbre gauche est vide ou sa racine est inférieure à la racine de a, et de même pour le sous-arbre droit.

D'où la fonction suivante, faisant usage d'une fonction `racine` renvoyant la racine d'un arbre non vide.

```
let racine a=match a with
  | Vide -> failwith "vide"
  | N(_,x,_) -> x
;;

let check_tpe a=
  let rec aux a=match a with
    | Vide -> true, 0
    | N(g,x,d) -> let b,n=aux g and c,m=aux d in b && c && (g=Vide || x>=racine g) &&
            (d=Vide || x>=racine d) && n-1<=m && m<=n, n+m+1
  in fst (aux a)
;;
```

Question 15. Pour l'insertion de x dans un tas t (non nécessairement équilibré), il suffit de suivre le principe suivant :

- si t est vide, on renvoie un tas contenant seulement x ;
- sinon, si x est inférieur à la racine r de t, on peut insérer arbitrairement dans le sous-arbre gauche ou droit de t ;
- dans le cas où x est strictement supérieur à la racine r de t, on remplace r par x, et on insère r dans le sous-arbre gauche ou droit.

La difficulté ici est de préserver la propriété d'arbre presque équilibré. L'astuce consiste à échanger les sous-arbres gauche et droit lorsqu'on insère récursivement. En effet, si t est un tas dont les sous-arbres gauche et droit sont g et d, alors $\mathcal{N}(d) \leq \mathcal{N}(g) \leq \mathcal{N}(d) + 1$. Si on insère un élément dans d pour obtenir d', on a alors $\mathcal{N}(g) \leq \mathcal{N}(d') \leq \mathcal{N}(g) + 1$: l'arbre ayant pour sous-arbre gauche d' et pour sous-arbre droit g vérifie la propriété d'APE en la racine !

```
let rec inserer a x=match a with
  | Vide -> N(Vide, x, Vide)
  | N(g,y,d) when y>x -> N(inserer d x, y, g)
  | N(g,y,d) -> N(inserer d y, x, g)
;;
```

La complexité est linéaire en la hauteur de l'arbre, donc en $O(\log n)$.

Question 16. On suit le principe de la descente de nœud du cours, même si ici l'implémentation est totalement différente puisque la structure n'est pas la même. On utilise la fonction `racine` écrite précédemment, en prenant bien garde de ne pas tenter d'accéder à la racine d'un arbre vide.

```
let rec modifier_racine a r=match a with
  | Vide -> Vide
  | N(g,_,d) when (g=Vide || racine g<=r) && (d=Vide || racine d<=r) -> N(g,r,d)
  | N(g,x,d) when (d=Vide || racine d<=racine g) -> N(modifier_racine g r, racine g, d)
  | N(g,x,d) -> N(g,racine d, modifier_racine d r)
;;
```

La complexité est linéaire en la hauteur de l'arbre, donc en $O(\log n)$.

Question 17.

```
let supprimer_racine a=
  let rec aux a=match a with
    | Vide -> failwith "Vide"
    | N(Vide, e, _) -> Vide, e
    | N(g,x,d) -> let g2,e=aux g in N(d,x,g2), e
  in let b,e=aux a in racine a, modifier_racine b e
;;
```

Là encore, la complexité est linéaire en la hauteur de l'arbre, donc en $O(\log n)$.

Voici pour l'exemple les arbres produits par insertions successives des éléments de $[\![0, 12]\!]$ (à gauche), puis le rajout de 13 (à droite).