
TP 2 : Arbres AVL, et tas presque équilibrés

1 Arbres AVL : implémentation

Télécharger le fichier `annexe_avl.ml`. Dans cette section, on implémente les principales fonctions sur les arbres AVL, qui sont des arbres binaires de recherche (ABR) presque équilibrés : la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit de tout noeud de l'arbre AVL est d'au plus 1. Pour des raisons d'efficacité dans les opérations, on utilise un champ supplémentaire par rapport aux ABR, indiquant la hauteur de l'arbre.

```
type 'a avl = Vide | N of int * 'a avl * 'a * 'a avl ;;
```

On convient qu'un arbre AVL vide est de hauteur -1 .

On dira qu'un arbre est un :

- quasi-AVL si c'est un arbre AVL, mais dont le champ hauteur (associé à la racine) est possiblement erroné ;
- presque-AVL si c'est un quasi-AVL ou si c'est un arbre dont les sous-arbres gauche et droit sont des AVL, dont les hauteurs diffèrent d'au plus 2.

1.1 Vérification des propriétés d'ABR

Question 1. Écrire une fonction `enum_infixe` `a` de type `'a avl -> 'a list` fournissant l'énumération infixé de l'arbre passé en entrée, sous la forme d'une liste.

```
#enum_infixe exemple_1 ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Question 2. Écrire une fonction `est_croissante_strict` `q`, de type `'a list -> bool` testant si la liste passée en paramètre est strictement croissante.

Question 3. En déduire une fonction `check_abr` de type `'a avl -> bool` testant si l'arbre passé en paramètre vérifie bien les propriétés d'arbre binaire de recherche.

```
#check_abr exemple_1 && check_abr exemple_2 && check_abr exemple_3 ;;
- : bool = true
```

1.2 Calcul de hauteurs dans des quasi-AVL

Question 4. Écrire une fonction `ha` `a` de complexité $O(1)$, de signature `'a avl -> int` retournant la hauteur de l'arbre AVL, fourni en entrée. On rappelle que `Vide` a pour hauteur -1 , et si l'arbre n'est pas vide, sa hauteur est accessible directement.

```
#ha exemple_3 ;;
- : int = 2
```

Question 5. Écrire une fonction `ha_bis` `a` de complexité $O(1)$, de signature `'a avl -> int` calculant la hauteur de l'arbre passé en entrée. On suppose que celui ci est un quasi-AVL. En déduire une fonction `recalc_ha` `a` de signature `'a avl -> 'a avl` et de complexité $O(1)$ prenant en entrée un quasi-AVL et renvoyant un AVL.

```
#ha_bis exemple_3 ;;
- : int = 3
#recalc_ha exemple_3 = exemple_1 ;;
- : bool = true
```

Question 6. Écrire une fonction `check_avl a`, de signature `'a avl -> bool` vérifiant que l'arbre passé en entrée est un arbre AVL correct (inutile de vérifier que c'est un ABR). On exige une complexité $O(n)$, avec n le nombre de noeuds (cette fonction ne sera utilisée dans la suite qu'à des fins de vérification).

```
#check_avl exemple_1 ;;
- : bool = true
#check_avl exemple_2 || check_avl exemple_3 ;;
- : bool = false
```

1.3 Rotations, Equilibrage, Insertion et Suppression

Le principe d'équilibrage d'un presque-AVL est basé sur les rotations, dont on rappelle le principe. Sur le schéma de la figure 1, x et y sont des noeuds, α , β et γ sont des arbres binaires de recherche. On suppose que les noeuds de α sont tous strictement inférieurs à x , qui est inférieur à tous les noeuds de β , eux-même inférieurs à y qui est plus petit que ceux de γ . En abrégé : $\alpha < x < \beta < y < \gamma$, de sorte que les deux arbres binaires sont des ABR corrects, avec les mêmes noeuds.

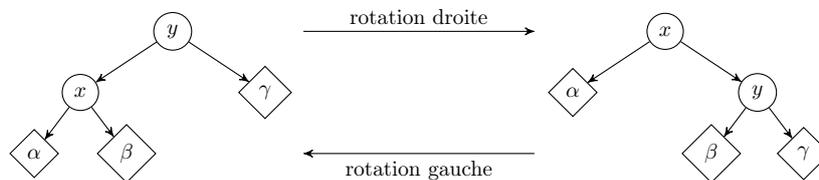


FIGURE 1: Rotations

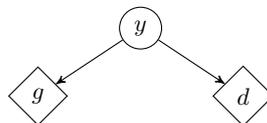
Question 7. Écrire une fonction `rotation_d a`, de complexité $O(1)$, de type `'a avl -> 'a avl`, telle que si a a la même forme que l'arbre de gauche (une racine et un sous-arbre gauche non vide), ses deux sous-arbres gauche et droite étant des arbres AVL (donc avec notamment α , β et γ des arbres AVL corrects), la fonction retourne l'arbre de droite, les hauteurs des nouveaux arbres enracinés en x et y ayant été calculées à l'aide de `recalc_ha`. On renverra une erreur si a n'a pas la forme nécessaire.

```
#rotation_d exemple_1 = exemple_2 ;;
- : bool = true
```

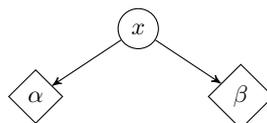
Question 8. Faire de même avec une fonction `rotation_g`.

```
#rotation_g exemple_2 = exemple_1 ;;
- : bool = true
```

Question 9. *Équilibrage d'arbres.* Cette fonction est la plus importante! Il s'agit d'écrire une fonction `equilibrer a`, prenant en entrée un presque-AVL et retournant un arbre AVL. Si a est non vide, il est de la forme



Si l'arbre total est un quasi-AVL, il suffit de recalculer sa hauteur. Sinon, il y a 4 cas à distinguer. En supposant $\text{hauteur}(g) = \text{hauteur}(d) + 2$ (on en déduit le cas $\text{hauteur}(d) = \text{hauteur}(g) + 2$ par symétrie), on le réduit à 2. Le sous-arbre g n'est donc pas vide, et s'écrit :



On distingue alors les deux cas suivants :

- si $\text{hauteur}(\alpha) \geq \text{hauteur}(\beta)$, alors l'arbre obtenu par rotation droite de l'arbre a est un arbre AVL ;
- sinon, l'arbre obtenu par rotation gauche de l'arbre g puis rotation droite de l'arbre total obtenu est un arbre AVL.

Implémenter la fonction `equilibrer a`.

1.4 Insertion et suppression dans un arbre AVL

Question 10. Écrire une fonction `max_abr` `a` de type `'a avl -> 'a` renvoyant le maximum de l'arbre `a`.

```
#max_abr exemple_1 ;;
- : int = 10
```

Question 11. Réécrire des fonctions d'insertion et de suppression d'un élément dans un arbre AVL. On suivra le même schéma que pour les ABR du cours, mais en appelant `equilibrer` « à la remontée » (après un appel récursif). Pour la suppression, on procédera comme dans le cours, en se ramenant à la suppression de la racine :

- si le nœud à supprimer ne possède pas de sous-arbre gauche, après suppression l'arbre associé est le sous-arbre droit ;
- sinon, on supprime d'abord le maximum m du sous-arbre gauche, et on crée un nouvel arbre AVL de racine m (penser à l'équilibrer).

1.5 Test

Question 12. Exécuter le script suivant, et vérifier que l'on obtient `true` :

```
let b=ref true ;;
let a=ref Vide ;;
for i=0 to 100 do
  a:= inserer !a i ;
  b:= !b && check_avl !a && check_abr !a
done ;
for i=0 to 100 do
  a:= supprimer !a i ;
  b:= !b && check_avl !a && check_abr !a
done ;
!b ;;
```

2 Tas presque équilibrés

Télécharger le fichier `annexe_tpe.ml`. Une définition de tas (max), plus générale que celle vue en cours, est la suivante : c'est un arbre binaire où l'étiquette d'un nœud est supérieure à celles de ses enfants, s'ils existent. Dans le cours, on s'est restreint aux tas complets à gauche, ce qui était naturel pour une implémentation dans un tableau. Le but de cette section est de détailler une implémentation de la structure de file de priorité avec des tas persistants. Bien sûr, on veut que les opérations soient le plus efficace possible, et donc maintenir des tas approximativement équilibrés. De manière surprenante, il est possible d'écrire facilement les opérations de file de priorité sur des tas persistants, tout en maintenant une hauteur logarithmique en le nombre de nœuds, sans stocker d'information supplémentaire comme pour les arbres AVL. L'idée est de ne travailler qu'avec des arbres *presque-équilibrés*.

On va pour cela travailler avec le type classique :

```
type 'a arbre = Vide | N of 'a arbre * 'a * 'a arbre ;;
```

Le nombre de nœuds $\mathcal{N}(a)$ d'un tel arbre est défini par induction : $\mathcal{N}(\text{Vide}) = 0$, et $\mathcal{N}(N(g,x,d)) = 1 + \mathcal{N}(g) + \mathcal{N}(d)$. La définition d'un arbre presque équilibré (en abrégé APE) est elle aussi inductive : un arbre est un APE s'il est vide, ou s'il est de la forme $N(g,x,d)$ avec g et d deux APE vérifiant $\mathcal{N}(g) - 1 \leq \mathcal{N}(d) \leq \mathcal{N}(g)$. On parle de tas presque équilibré (TPE) lorsque l'arbre satisfait de plus la propriété de tas.

Question 13. Montrer qu'un APE non vide à n nœuds a une hauteur $\lfloor \log_2(n) \rfloor$. (Pour rappel, la hauteur de l'arbre vide est -1).

Question 14. Écrire une fonction `check_tpe` `a` prenant en entrée un arbre, et vérifiant si c'est un TPE. On impose une complexité $O(n)$ avec n le nombre de nœuds. (Indication : utiliser une fonction auxiliaire. On pourra aussi écrire une fonction renvoyant la racine d'un arbre non vide).

```
#check_tpe exemple_4 ;;
- : bool = true
#check_tpe exemple_5 || check_tpe exemple_6 ;;
- : bool = false
```

Question 15. Écrire une fonction `insérer a x` de type `'a arbre * 'a -> 'a arbre`, prenant en entrée un TPE et un élément x , et renvoyant un nouveau TPE contenant les mêmes éléments ainsi que x . Réfléchir à comment maintenir la structure d'APE lorsqu'on ajoute un nœud (*indication* : dans la définition d'un tas (max), l'important est que le maximum soit en haut. Peu importe qui est à gauche et qui est à droite!) On impose une complexité $O(\log n)$ avec n le nombre de nœuds du TPE.

```
#insérer exemple_4 11 = exemple_7 ;;
- : bool = true
```

La question qui suit a pour but de faciliter la suppression de la racine.

Question 16. Écrire une fonction `modifier_racine a r` de type `'a arbre * 'a -> 'a arbre`, prenant en entrée un TPE a , et remplaçant sa racine par r . On prendra garde à maintenir la structure de tas : en pratique on fait descendre la nouvelle racine r jusqu'à obtenir un tas, comme dans le cours. On impose une complexité $O(\log n)$ avec n le nombre de nœuds du TPE.

```
#modifier_racine exemple_7 (-1) = exemple_8 ;;
- : bool = true
```

Question 17. Pour supprimer la racine d'un TPE a non vide, on procède de la façon suivante :

- on supprime le nœud le plus à gauche dans a et on permute les branches gauche et droite de chaque nœud ascendant du nœud supprimé. L'arbre obtenu est noté a' , et l'étiquette supprimée e .
- on remplace l'étiquette r de a' par e et on reconstitue un APE a'' via la fonction de la question précédente.

Écrire une fonction `supprimer_racine a` de type `'a arbre -> 'a * 'a arbre`, prenant en entrée un TPE et renvoyant sa racine et un TPE où cette racine a été supprimée. On impose une complexité $O(\log n)$ avec n le nombre de nœuds du TPE.

```
#supprimer_racine exemple_7 ;;
- : int * int arbre =
  11,
  N
  (N (N (N (Vide, 3, Vide), 7, Vide), 9, N (N (Vide, 1, Vide), 5, Vide)), 10,
   N (N (N (Vide, 2, Vide), 6, Vide), 8, N (N (Vide, 0, Vide), 4, Vide)))
```