Algorithmique et Structures de données

L2 2021-2022

Travaux Pratiques 8 - Arbre AVL -

`gcc mon_fichier.c -std=c11 -Wall -Wextra -o mon_programme`
Les exercices marqués d'un @ sont optionnels et à faire dans un second temps.

On utilisera les définitions de types suivantes pour les arbres AVL :

```
typedef struct nAVL
{
   int val;
   int h; // h mémorise la hauteur du sous-arbre du noeud
   struct nAVL* g; // pour fils gauche
   struct nAVL* d; // pour fils droit
} NAVL; // noeud d'un arbre AVL
typedef NAVL* AVL; // Arbre AVL
AVL a= NULL; // l'arbre vide
```

Exercice 1. *Ajouter*

On souhaite fabriquer une fonction pour ajouter des éléments à l'arbre tout en conservant sa propriété d'arbre AVL. On commence par construire des sous-fonctions.

- `void MAJHauteur(NAVL* no)` : La fonction met à jour le champs `h` du noeud `no` en fonction de la hauteur de ses deux noeuds fils (attention aux fils vides). La fonction ne doit pas recalculer toute la hauteur mais se fier aux valeurs des champs hauteurs des deux fils le cas échéant.
- `void rotationG(AVL* ar)` et `void rotationD(AVL* ar)` : Les fonctions effectuent les rotations gauche et droite de l'arbre `*ar`. La fonction met à jour les champs `h`.
- `void equilibrer(AVL* ar)` : La fonction qui équilibre `*ar` s'il est déséquilibré (par exemple à la suite d'un **seul** ajout où d'une **seule** suppression). Elle doit maintenir la cohérence des champs `h`.
- `void ajouter(AVL* ar,int x)` : La fonction ajoute l'entier `x` à l'arbre `*ar`. Si l'entier existe déjà on ne fait rien. La fonction doit éventuellement rééquilibrer l'arbre après l'ajout, elle doit aussi conserver la cohérence des champs `h` et la structure d'ABR. Aide : les fonctions précédentes peuvent être utiles.

Exercice 2. *Recherche*

Écrire la fonction `NAVL* rechercher(AVL ar,int x)` qui renvoie l'adresse du noeud si l'entier `x` est présent dans l'arbre `ar`. Si `x` n'est pas dans `ar` renvoie `NULL`.

Exercice 3. *Vider*

Écrire la fonction `void vider(AVL* ar)` qui vide l'AVL `*ar` et libère la place occupée par les noeuds.

Exercice 4. *Extraction*

1. Écrire la fonction `NAVL* extraireMax(AVL* ar)` qui extrait la valeur maximum de l'arbre `*ar`. La fonction doit éventuellement faire des rotations pour conserver un arbre équilibré, elle doit aussi conserver la cohérence des champs `h` et la structure d'AVL.
2. Écrire la fonction `NAVL* extraire(AVL* ar,int x)` qui extrait la valeur `x` de l'arbre `*ar`. Si la valeur `x` n'est pas présente la fonction ne fait rien. La structure AVL et la cohérence des champs `h` doivent être conservées.

Exercice 5. *@Balance*

On voudrait améliorer en espace nos AVL. Pour cela on remplace le champs hauteur par le champs `bal` (balance) qui mémorise la différence de hauteur entre le fils gauche et le fils droit : $h_g - h_d = \boxed{\texttt{bal}}$. (On pourrait faire tenir ce champs sur 2 bits)

```
typedef struct nBal
{
   int val;
   char bal; // bal pour balance
   struct nBal* g; // pour fils gauche
   struct nBal* d; // pour fils droit
} NBal;
typedef NBal* ABal; // AVL avec balance
```

Écrire les fonctions :

— `int ajouterB(ABal* ar,int val)` qui renvoie la différence de hauteur (voir suite).
— `int rechercherB(ABal* ar,int val)` qui renvoie 1 si val est dans `*ar`, 0 sinon.
— `int supprimerB(ABal* ar,int val)` qui renvoie la différence de hauteur (voir suite).
— `void viderB(ABal* ar)`.

Afin de pouvoir mettre à jour les champs balances, les fonctions récursives `ajouterB` et `supprimerB` doivent renvoyer la différence entre la hauteur après l'action et la hauteur avant l'action. Elles doivent mettre à jour les champs balances en conséquence. On a les relations :

— $\boxed{\text{balance}} = h_{\text{sous-arbre de gauche}} - h_{\text{sous-arbre de droite}}$
— $\boxed{\text{valeur renvoyée par } \texttt{ajouterB} \text{ ou } \texttt{supprimerB}} = h_{\text{apres}} - h_{\text{avant}}$.

où :

— h_{avant} : hauteur de l'arbre avant appel de `ajouterB` ou `supprimerB` sur l'arbre.
— h_{apres} : hauteur de l'arbre après appel de `ajouterB` ou `supprimerB` sur l'arbre.

Exercice 6. *@Dictionnaire*

On veut maintenant implanter un dictionnaire utilisant la structure d'AVL. On ajoute un champs de type `void*` afin d'associer un objet à chaque valeur entière.

```
typedef struct dico
{
   int val;
   char bal; // bal pour balance
   void* obj; // obj pour objet
   struct dico* g; // pour fils gauche
   struct dico* d; // pour fils droit
} Ndico; //
typedef Ndico* Dico; // dictionnaire
Dico a= NULL; // dictionnaire vide.
```

Implanter les fonctions suivantes :

— `Ndico* ajouterD(Dico* dict,int val)` qui ajoute la valeur `val` dans le dictionnaire `*dict`. Si la valeur `val` est déjà présente ne construit rien. Dans tous les cas, renvoie simplement l'adresse du noeud associé à la valeur `val`. Un utilisateur pourra modifier le champs `obj` du noeud grâce à la fonction `void affObjN(Ndico* no,void* obj)`.
— `void affObjN(Ndico* no,void* obj)` qui affecte le champs du noeud `no` avec la valeur `obj`.
— `void* getObjN(Ndico* no)` qui renvoie l'adresse de l'objet du noeud (Renvoie la valeur du champs `obj` du noeud).
— `Ndico* getNoD(Dico* dict,int val)` qui renvoie l'adresse du noeud associé à la valeur `val` si cette dernière est présente dans `dict`. Sinon renvoie `NULL`.
— `void* extraireD(Dico* dict,int val)` qui supprime la valeur `val` de `dict` et libère la mémoire occupée par le noeud associé. Renvoie l'objet associé à la valeur `val`.
— `void vider(Dico* dict)` qui vide le contenu du dictionnaire et libère la mémoire associée aux noeuds. Attention la fonction ne doit pas libérer la mémoire associée aux objets du dictionnaire.

Exercice 7. *@Perfectionnement*

Écrire `Ndico* ajoutCondD(Dico* dict,int val)`. Si la valeur `val` n'était pas présente dans le dictionnaire, cette fonction l'ajoute et renvoie l'adresse du noeud construit. Sinon renvoie la valeur `NULL`.