Algorithmique et Structures de données 1

L2 2021-2022

Travaux Pratiques 8

Site du cours : `https://defelice.up8.site/algo-struct.html`
Les exercices marqués de (@) sont à faire dans un second temps.
Un fichier écrit en langage C se termine conventionnellement par `.c`.
Une commande de compilation est `gcc fichier_source1.c fichier_source2.c fichier_source3.c`.
Voici des options de cette commande.

- `-o nom_sortie` pour donner un nom au fichier de sortie (par défaut `a.out`).
- `-Wall -Wextra` pour demander au compilateur d'afficher plus de Warnings
- `-std=c11` pour compiler selon la norme C11
- `-g -fsanitize=address` pour compiler avec information de débogage et en interdisant la plupart des accès a une zone mémoire non reservée.

Exemple : `gcc -Wall fichier1.c -o monprogramme`

La suite du TP utilise `noeudh_t` : une structure d'arbre binaire étiqueté avec mémorisation de la hauteur. Voici la définition de `noeudh_t` :

```
typedef struct s_noeudh_t
{
   int v; // étiquette du noeud (v pour valeur)
   struct s_noeudh_t* g; // pointeur le sous-arbre gauche
   struct s_noeudh_t* d; // pointeur le sous-arbre droit
   int h; // hauteur du sous-arbre enraciné en ce noeud (h pour hauteur)
} noeudh_t;
```

1 Arbre binaire de recherche simple (non équilibré)

Un arbre binaire de recherche (ABR) est un arbre qui vérifie que pour chacun de ses noeuds s

- les noeuds descendants de s à gauche ont des étiquettes strictement inférieures à celle de s
- les noeuds descendants de s à droite ont des étiquettes strictement supérières à celle de s

Dans un premier temps on ne s'occupe pas du champs `hauteur`. Les fonctions suivantes travaillent sur de simples arbres binaires de recherche pas forcément équilibré.

Exercice 1. *@Ajout*

Écrire la fonction `int ajouterABR(noeudh_t** a,int x)` qui ajoute l'entier `x` à l'arbre `*a`. Si l'entier `x` est déjà présent on ne modifie pas l'arbre. Renvoi 1 si l'arbre a été modifié, 0 sinon.

Exemple : `noeudh_t* arbre=NULL; ajouterABR(&arbre,3);`

Exercice 2. *@Recherche*

Écrire la fonction `int chercher(noeudh_t* a,int x)` qui renvoie 1 si l'entier `x` est présent dans l'arbre `a`. Si `x` n'est pas dans `a` renvoie `0`.

Exercice 3. *@Extraire max*

Écrire la fonction `int extraireMaxABR(noeudh_t** a)` qui retire de l'arbre non vide `*a` le noeud ayant la plus grande étiquette et renvoie la valeur extraite. Pensez à desallouer la place occupée pour le noeud extrait.

Exercice 4. *@Extraire min*

Écrire la fonction `extraireMin`.

Exercice 5. *@Supprime*

Écrire la fonction `int supprimerABR(noeudh_t** a,int x)` qui supprime de l'arbre le noeud ayant pour valeur `x` s'il est présent. renvoi 1 si l'arbre a été modifié.

On dira qu'un arbre est un équilibré si pour chaque noeud s le sous-arbre de droite et le sous-arbre de gauche ont des hauteurs égale à 1 près. Pour la hauteur on prendra comme convention : l'arbre vide est de hauteur -1.

On appellera AVL un arbre de recherche qui est équilibré.

A partir de maintenant on souhaite réécrire les fonctions d'ajout et de suppression d'une étiquette dans l'arbre mais

- On suppose que avant l'ajout ou la suppression, l'arbre est un AVL et doit le rester après. Il faut pour cela le rééquilibrer si besoin par une ou plusieurs rotations après chaque modification.
- Afin de savoir si une partie de l'arbre a besoin d'être équilibrée on utilise le champs `h` situé dans chaque noeud et qui contient la hauteur du sous-arbre dont la racine est le noeud. On suppose que ce champs est correcte et il faut le mettre à jour à chaque modification (rotation comprise).

2 AVL, fonctions intermédiaires

On commence par écrire des sous-fonctions dont on aura besoin dans `ajouterAVL` et `supprimerAVL`. Vous pouvez éventuellement ne pas suivre la stratégie proposée et écrire les fonctions `supprimerAVL` et `ajouterAVL` à votre idée.

Exercice 6. *@Hauteur*

Écrire la fonction `int hauteurAVL(noeudh_t* r)` qui renvoie la hauteur de l'arbre `r` en utilisant le champs `h` du noeud `r` si `r` est non vide.

Exercice 7. *@Mise à jour Hauteur*

Écrire la fonction `void miseAJHaut(noeudh_t* r)` qui met à jour le champs `h` du noeud `r` 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.

Exercice 8. *@Rotation*

Écrire `void rotationG(noeudh_t** r)` (et `void rotationD(noeudh_t** r)`). Ces fonctions effectuent les rotations gauche (et droite) de l'arbre sur le noeud `r`. On suppose que la structure de l'arbre enraciné en `*r` s'y prête. La fonction met à jour les champs `h`.

Exercice 9. *@Équilibre*

Écrire la fonction `void equilibrer(AVL* r)` fonction qui équilibre `*r` grâce à des rotations, si `*r` est déséquilibré (par exemple à la suite d'un **seul** ajout où d'une **seule** suppression). La fonction doit doit maintenir la cohérence des champs `h`.

3 AVL, fonctions principales

Voici les fonctions principales, à priori elles utilisent les fonctions précédentes et opèrent sur des AVL avec des valeurs de champs `h` cohérents. Après chaque opération la cohérence des `h` et la structure d'AVL est conservée.

Exercice 10. *Ajout*

Écrire la fonction `int ajouterAVL(noeudh_** r,int x)`. La fonction ajoute l'entier `x` à l'arbre `*r`. Si l'entier est déjà présent dans `*r` la fonction ne fait rien et renvoie 0. En cas d'ajout la fonction doit renvoyer 1.

Exercice 11. *Extraction max*

Écrire la fonction `int extraireMaxAVL(noeudh_t** ar)` qui extrait la valeur maximum de l'arbre non vide `*r` et la renvoie.

Exercice 12. *@Extraction min*

Écrire la fonction `int extraireMinAVL(noeudh_t** ar)`.

Exercice 13. *Suppression*

Écrire la fonction `int supprimerAVL(noeudh_t** r,int val)` qui supprime le noeud d'étiquette `val` si elle est présente dans `*r`. Renvoie 1 si effectivement un noeud a été supprimé 0 sinon.