Cours de programmation séquentielle

Arbres AVL

1 Objectifs

- Compréhension et implémentation des arbres AVL,
- Utilisation de la récursivité,
- Manipulation des pointeurs.

2 Préambule

Ce travail pratique reprend l'implémentation des arbres binaires de recherche et lui ajoute la notion d'équilibrage AVL (voir les slides du cours 17 et bientôt du cours 18).

Dans un premier temps, on s'occupe d'avoir un arbre binaire de recherche pour lequel l'insertion est implémentée avec la récursivité. Les nombres insérés dans l'arbre sont uniques. On doit donc aussi disposer d'une fonction de recherche.

Ensuite, on calcule les hauteurs des sous-arbres gauche et droite des nœuds de l'arbre. On repère, le cas échéant, le premier nœud déséquilibré en remontant le chemin d'insertion et on rééquilibre son sous-arbre en opérant une rotation simple ou double selon le type de déséquilibre. Finalement, on recalcule les hauteurs du sous-arbre rééquilibré. Dans un deuxième temps, on optimisera l'insertion en gérant directement à partir de la fonction d'insertion la mise à jour des hauteurs, lorsqu'on remonte le chemin d'insertion, et le rééquilibrage.

On passera en dernier lieu à l'implémentation de la suppression seulement si l'insertion fonctionne de manière optimisée.

On utilisera la structure de donnée suivante pour l'arbre AVL :

```
typedef struct _node {
    int key;
    struct _node* left, right; //sous-arbres gauche et droite
    int lh, rh; // hauteurs des sous-arbres gauche et droite
} node;

typedef node* avl_tree;
```

3 Travail à réaliser

- Étudier dans le détail les notions de hauteurs et d'équilibre AVL pour un arbre binaire de recherche. Construire des exemples !
- Écrire une fonction récursive `avl_insert()` qui insère correctement une valeur dans l'arbre.
- Écrire une fonction `avl_search()` qui vérifie si une valeur se trouve dans l'arbre.
- Écrire une fonction `avl_compute_height()` qui calcule la hauteur d'un arbre.
- Écrire une fonction récursive `avl_compute_heights()` qui calcule les hauteurs des sous- arbres gauche et droite des nœuds de l'arbre.
- Écrire une fonction `avl_is_unbalanced()` qui détermine, le cas échéant, le premier nœud déséquilibré en remontant le chemin d'insertion (donc en partant de l'élément inséré en direction de la racine).
- Écrire une fonction `avl_rebalance()` qui rééquilibre si nécessaire l'arbre après une insertion.
  - Cette fonction opère une ou deux rotations (gauche ou droite) à partir du nœud déséquilibré selon la configuration autour de ce nœud. Il est donc nécessaire d'implémenter des fonctions `avl_left_rotation()`, `avl_right_rotation()`, `avl_left_right_rotation()`, `avl_right_left_rotation()`.
  - Avant de programmer cette fonction, étudier dans le détail les configurations possibles
- Le programme principal permettra d'insérer successivement des éléments dans l'arbre, en maintenant celui-ci équilibré AVL après chaque insertion. L'arbre sera affiché avec la fonction `avl_print()` après chaque insertion.
- Une fois le travail ci-dessus réalisé, modifier la fonction `avl_insert()` afin de ne recalculer que les hauteurs susceptibles d'avoir été modifiées après rééquilibrage.
- Passer à la suppression en implémentant la fonction `avl_delete()` qui supprime une valeur dans l'arbre.
  - Il faut d'abord implémenter la suppression pour un arbre binaire de recherche avant de rajouter les rééquilibrages successifs en remontant le chemin à partir du nœud supprimé jusqu'à la racine.
  - La remontée nécessite de mettre à jour les hauteurs en remontant le chemin pour voir si un rééquilibrage est nécessaire.

4 Bonus

Comparer les performances de l'insertion, de la recherche et de la suppression entre les arbres de recherche binaires et ceux équilibrés AVL.

- Il s'agit de mesurer les temps moyens de ces opérations en fonction de la taille n de l'arbre où n est le nombre de nœuds.
- Prendre des valeurs de n assez grandes et tracer des graphiques.