L2 Informatique

TP Algorithmique – Arbres binaires de recherche

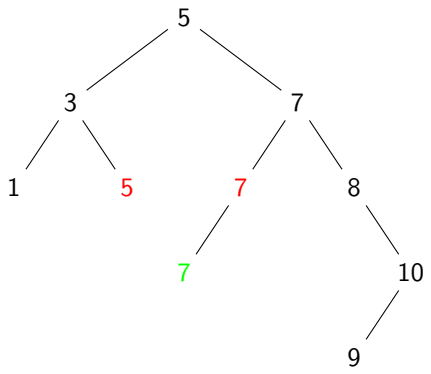
L'objectif de ce TP est d'implémenter un certain nombre de fonctions sur les arbres binaires de recherche (ABR). On considèrera des ABR où l'étiquette d'un nœud contient une valeur entière. Les ABR que l'on va traiter peuvent contenir plusieurs fois la même étiquette.

Partie I

1. Définir le type `Noeud` et le type `Arbre` permettant de représenter des ABR contenant des entiers (représentation à l'aide de **pointeurs**).

2. Écrire un sous-programme `Ajouter` qui insère un entier `e` comme une feuille dans un ABR.

3. Écrire un sous-programme `Generer` qui un crée un nouvel ABR et lui ajoute, dans l'ordre, l'ensemble des valeurs contenues dans un tableau T comportant n entiers.
   Par exemple, en considérant $T_1 = [5, 3, 7, 1, 8, 10, 9, 5, 7, 7]$ et $n_1 = 10$, `Genere(T1,n1)` devra retourner l'arbre suivant[1] :



L'ABR a_1 construit à partir de T_1.

4. Écrire un sous-programme d'affichage d'un ABR selon un parcours infixe. Chaque ligne comportera les informations sur un nœud : élément (valeur du nœud), adresse du nœud (les valeurs des pointeurs peuvent être affichées), adresse des sous-arbres gauche et droit. Les éléments de l'arbre devraient être affichés dans l'ordre croissant, soit 1, 3, 5, 5, 7, 7, 7, 8, 9, 10 pour l'arbre de l'exemple ci-dessus.

5. Écrire un sous-programme de recherche d'un entier dans un ABR. Il s'agit d'une fonction booléenne retournant `vrai` si et seulement si l'élément recherché `e` appartient à un ABR `a`.

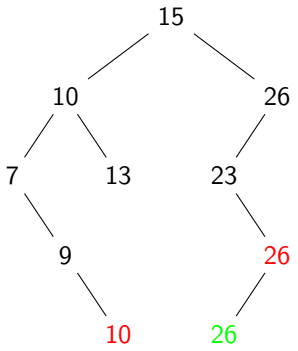
6. Écrire un sous-programme de suppression d'un entier `e` dans un ABR `a`. Si `e` n'appartient pas à `a`, l'ABR restera inchangé. Si `e` apparait plusieurs fois dans `a`, on supprimera le premier rencontré.

7. Écrire une fonction calculant la somme des valeurs d'un ABR `a` strictement inférieures à un entier `x` donné. Il est demandé d'optimiser le calcul en explorant seulement les nœuds utiles de `a`. Si aucun élément de l'arbre n'est inférieur à `x`, on retournera 0.

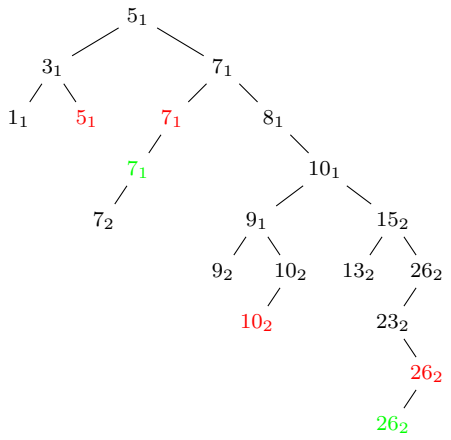
8. Écrire un sous-programme `Fusionner` qui, étant donnés deux ABR a_1 et a_2, ajoute dans a_1 tous les éléments de a_2 (considérés dans l'ordre d'un parcours préfixe).
   La figure ci-après montre les modifications apportées à a_1 après fusion avec un arbre a_2 généré à partir du

[1] Les couleurs permettent de distinguer les entiers de même valeur. Par exemple, la valeur de la racine de l'arbre correspond au premier 5 de T_1.

tableau [15, 10, 26, 7, 9, 23, 13, 26, 10, 26]. Sur cette illustration d'arbres fusionnés, "e_i" signifie que e provient de a_i.



ABR a_2



a_1 après fusion avec a_2

9. Écrire un sous-programme qui teste si un arbre binaire est équilibré, c'est-à-dire dont tout sous-arbre est tel que la différence de hauteur entre son sous-arbre gauche et son sous-arbre droit (en valeur absolue) est de 1 au maximum. Par exemple, les arbres a_1 (avant et après fusion) et a_2 ne sont pas équilibrés, au contraire de l'ABR généré à partir du tableau [11, 4, 15, 3, 7, 17, 6].

10. Écrire un sous-programme qui libère la mémoire allouée pour stocker un ABR en utilisant un parcours postfixe. On affichera les emplacements successivement libérés (valeurs et adresse des nœuds) pour vérification.
Un parcours postfixe de l'arbre a_2 libèrerait dans l'ordre : 10, 9, 7, 13, 10, 26, 26, 23, 26, 15.

Partie II : Arbres AVL

11. Ajouter dans le type `Noeud` un champ `hauteur` permettant de stocker la hauteur de l'arbre (ou du sous-arbre) dont ce nœud est la racine. Pour simplifier les traitements à suivre, écrire une fonction retournant la hauteur d'un arbre (valeur du champ hauteur si l'arbre est non vide, ou -1 sinon) et un autre sous-programme permettant de mettre à jour la hauteur du nœud racine d'un arbre à partir des hauteurs de ses sous-arbres.

12. Écrire les 4 opérations de rotation d'arbres AVL : gauche, droite, gauche-droite et droite-gauche.

13. Écrire un sous-programme premettant de `Reequilibrer` un arbre `a`, à savoir vérifier le facteur d'équilibre du nœud racine, appliquer le rééquilibrage le cas échéant, et mettre à jour les champs `hauteur`. On suppose que les sous-arbres de `a` sont déjà équilibrés.

14. Écrire l'opération `AjouterAVL` qui insère un entier `e` dans un arbre supposé équilibré en employant le principe des arbres AVL : l'ajout s'effectue de la même manière que pour un ABR classique, avec éventuel rééquilibrage de l'arbre et mise à jour des champs `hauteur`.

15. Écrire un sous-programme `GenererAVL` qui crée un nouvel arbre AVL en lui ajoutant, dans l'ordre, l'ensemble des valeurs contenues dans un tableau T (similairement à la question 3 de la partie I).

16. Dans le programme principal, effectuer une comparaison entre deux 2 ABR générés au moyen d'un même tableau de nombres, selon chacune des stratégies d'ajout (classique ou AVL) : affichage infixe des valeurs, hauteur des arbres (pour établir la hauteur d'un ABR classique, écrire au préalable un sous-programme dédié) et vérification de leur équilibrage.

17. Enrichir le programme afin de comparer les hauteurs des arbres de plus grande taille, en générant des tableaux de valeurs pouvant conduire à des ABR classiques très déséquilibrés.