

TP 8 : Arbres binaires de recherche

Semaine du 19 mars 2007

Nous allons implémenter des arbres binaires de recherche (*ABR* dans la suite) et les opérations pour les manipuler. Un *ABR* est tout d'abord un arbre « binaire » : chacun de ses noeuds comporte au plus deux fils. On attache à chaque noeud une valeur (entière dans ce TP). Enfin un *ABR* est construit de manière à rendre facile la recherche de valeur parmi celles qu'il contient. Pour cela, on définit les règles (inductives) suivantes :

1. toutes les valeurs *strictement inférieures* à celle de la racine sont placées dans les noeuds du sous-arbre *gauche* de la racine.
2. toutes les valeurs *supérieures* ou *égales* à celle de la racine sont placées dans les noeuds du le sous-arbre **droit** de la racine.
3. les sous-arbres gauche et droit de la racine sont aussi des *ABR*.

La figure 1 montre un exemple d'*ABR* contenant des entiers.

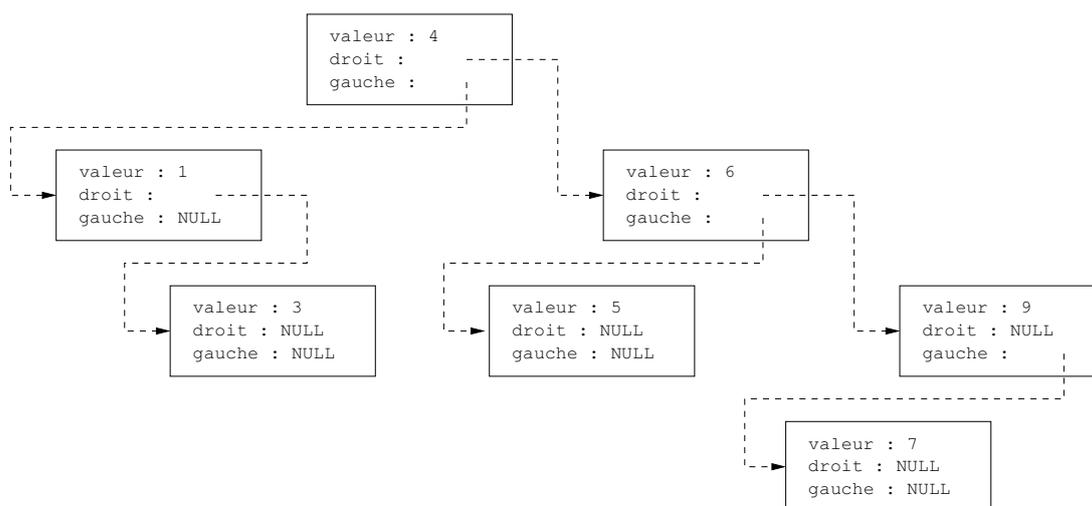


FIG. 1 – Un arbre binaire de recherche

Remarques :

- durant tout ce TP, le détail des structures à implémenter ainsi que les prototypes exacts des fonctions ne seront pas donnés dans l'énoncé. Vous êtes donc libres d'utiliser l'implémentation que vous préférez (mais il n'y a pas 36 possibilités et il vaut mieux faire simple...),
- n'oubliez pas de vérifier régulièrement sur des exemples le bon fonctionnement des fonctions que vous écrivez.

Exercice 1 Définir une structure `struct noeud_s` permettant de coder un noeud d'un arbre binaire contenant une valeur entière. Ajouter des `typedef` pour définir les nouveaux types `noeud_t` et `arbre_t` (ces types devraient permettre de représenter une feuille, c'est à dire un arbre vide, un noeud sans valeur attachée ni fils).

Exercice 2 Écrire une fonction `Cree_arbre()` qui prend en argument une valeur entière ainsi que deux arbres et renvoie un arbre dont la racine contient cette valeur et les deux sous-arbres sont ceux donnés en paramètre.

Exercice 3 Écrire une fonction (récursive) `detruit_arbre()` qui libère la mémoire occupée par tous les noeuds d'un arbre binaire.

Exercice 4 Écrire une fonction (récursive) `nombre_de_noeuds()` qui calcule le nombre de noeuds d'un arbre binaire.

Exercice 5 Écrire une fonction `affiche_arbre()` qui affiche les valeurs des noeuds d'un *ABR* par ordre croissant (choisissez le bon type de parcours des noeuds de l'arbre...).

Exercice 6 Écrire une fonction `affiche_arbre2()` permettant d'afficher les valeurs des noeuds d'un arbre binaire de manière à lire la structure de l'arbre. Un noeud sera affiché ainsi : $\{g, v, d\}$ où g est le sous-arbre gauche, v la valeur du noeud et d le sous-arbre droit. Par exemple, l'arbre de la figure 1 sera affiché par : $\{_, 1, \{_, 3, _ \}\}, 4, \{\{_, 5, _ \}, 6, \{\{_, 7, _ \}, 9, _ \}\}$. Les ' _ ' indiquent les sous-arbres vides.

Exercice 7 Écrire une fonction `compare()` qui compare deux arbres binaires (la fonction renvoie une valeur nulle si et seulement si les deux arbres binaires ont la même structure d'arbre et qu'ils portent les mêmes valeurs aux noeuds se correspondant).

Exercice 8 Écrire une fonction (récursive...) `insere()` qui ajoute une valeur dans l'*ABR* (ce sera un nouveau noeud placé correctement dans l'arbre).

Exercice 9 Écrire une fonction `trouve_noeud()` qui renvoie l'adresse d'un noeud de l'*ABR* donné en paramètre contenant une certaine valeur (ou NULL si cette valeur ne figure pas dans l'arbre).

Exercice 10 — (*difficulté* : ●) Écrire une fonction `verifie()` qui renvoie un entier non nul si et seulement si l'arbre binaire passé en paramètre est un arbre binaire de recherche. *Remarque* : on pourra écrire une fonction auxiliaire (récursive) qui vérifie qu'un arbre binaire (non vide) satisfait les propriétés d'*ABR* et en même temps détermine les valeurs minimales et maximales contenues dans cette arbre binaire (et les renvoie via des pointeurs...).

Exercice 11 — (*difficulté* : ●●) Écrire une fonction `tri()` qui trie un tableau d'entiers donné en argument à l'aide d'un arbre binaire de recherche. *Remarque* : on pourra écrire une fonction auxiliaire récursive qui, à partir d'un sous-arbre d'un *ABR* et d'une position dans le tableau, remplit le tableau, à partir de la position donnée, avec les valeurs contenues dans ce sous-arbre binaire et renvoie le nombre de valeurs du sous-arbre...

Exercice 12 — *Bonus* (*difficulté* : ●●●) Écrire une fonction `supprime()` qui supprime une valeur de l'arbre (on supprimera la première rencontrée) tout en conservant les propriétés d'*ABR*. L'algorithme est le suivant (une fois trouvé le noeud contenant la valeur en question) :

- si le noeud à enlever ne possède aucun fils, on l'enlève,
- si le noeud à enlever n'a qu'un fils, on le remplace par ce fils,
- si le noeud à enlever a deux fils, on le remplace par le sommet de plus petite valeur dans le sous-arbre droit, puis on supprime ce sommet.