Travaux Pratiques d'algorithmique n°9

Cours d'Informatique de Deuxième Année

—Deug SSM M3-MI—

Arbres binaires de recherche

L'objet de cette feuille d'exercices est l'implémentation des opérations de base sur une nouvelle structure de données récursive : la structure d'*arbre binaire de recherche*. L'algorithmique que l'on utilise sur ces structures est voisine de celle que l'on utilise sur les listes chaînées à ceci près que l'on ne parle plus du "suivant" d'une cellule mais plutôt du "fils gauche" et "fils droit" d'un nœud. De même, on ne parle plus du "début" de la liste mais plutôt de la "racine" de l'arbre.

---

1 Structure d'ABR et algorithmes de base

On considère les déclarations suivantes :

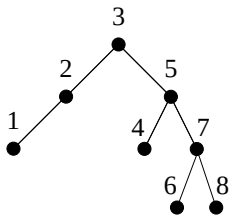
```
typedef struct noeud
{ float Val;               /* valeur dans le noeud                */
  int NbOcc;               /* nombre d'occurences de cette valeur */
  struct noeud * Gauche;   /* pointeur sur le sous-arbre gauche   */
  struct noeud * Droit;    /* pointeur sur le sous-arbre droit    */
} Noeud;                   /* nouveau type : Noeud                */

typedef Noeud * ArbreBin;  /* pointeur sur un noeud               */
```

Un arbre binaire, représenté par la structure `ArbreBin` ci-dessus, est dit *arbre binaire de recherche* ou encore *ABR* si les conditions suivantes sont vérifiées :

- ▷ l'arbre vide est un *ABR* ;
- ▷ les deux fils de la racine de l'arbre sont des *ABR* et la valeur à la racine de l'arbre est strictement plus grande (resp. petite) que toutes les valeurs du sous-arbre gauche (resp. droit).

Exemple :



▸ **Exercice 1. (Fonctions de base)** *Afin de pouvoir manipuler une telle structure de données, nous demandons d'écrire les fonctions élémentaires suivantes :*

1. `Noeud * CreerFeuille(float val, int nbocc)` *qui retournera un pointeur sur un nœud contenant* `nbocc` *occurence(s) de la valeur* `val` *;*
2. `int EstFeuille(ArbreBin abr)` *qui testera si le nœud pointé par* `abr` *est une feuille ;*
3. `int EstVide(ArbreBin abr)` *qui testera si l'arbre pointé par* `abr` *est vide.*

▸ **Exercice 2. (Recherche)** *Donner une fonction de nom* `RechercheBin` *prenant en paramètre un entier* `x` *et un arbre binaire de recherche pointé par* `abr` *et retournant le nombre d'occurences de l'élément* `x` *dans l'arbre pointé par* `abr`*.*

▸ **Exercice 3. (Parcours)** *Donner les fonctions de parcours préfixe, suffixe et infixe, d'un arbre binaire de recherche.*

2 Algorithmes complexes

▸ **Exercice 4. (Insertion)** *On considère l'opération d'insertion d'un élément dans un arbre binaire de recherche. Comme pour les listes chaînées triées, cette opération se caractérise par la propriété suivante : si* `abr` *est un arbre binaire de recherche et* `x` *un élément, alors le résultat de l'insertion de* `x` *dans l'arbre* `abr` *est encore un arbre binaire de recherche.*

1. *nous demandons tout d'abord de construire à la main les arbres binaires de recherche correspondant aux insertions successives dans l'arbre vide, des valeurs* $3, 5, 7, 2, 4, 1$ *;*
2. *déduire des exemples précédents l'algorithme d'insertion d'un élément* `x` *dans un arbre binaire de recherche pointé par* `abr`*;*
3. *donner alors la fonction de nom* `InsertionBin` *prenant en paramètre* `x` *et* `abr` *réalisant cet algorithme ;*
4. *évaluer la complexité.*

▸ **Exercice 5. (Utilisation d'un arbre binaire de recherche pour réaliser un tri)** *Ecrire une fonction de nom* `TriBin` *de tri d'un tableau d'entiers* `t` *à* `n` *éléments. Cette fonction utilisera un arbre binaire de recherche comme structure intermédiaire. Le résultat du tri sera stocké dans ce même tableau* `t`*.*

▸ **Exercice 6. (Suppression)** *Donner une fonction de nom* `SuppressionBin` *de suppression d'un élément* `x` *dans un arbre binaire de recherche pointé par* `abr`*. La fonction supprimera une seule occurence de* `x`*. Le résultat de cette suppression sera bien évidemment encore un arbre binaire de recherche.*