TP 8 : Arbres binaires de recherche

Semaine du 17 Mars 2008

Exercice 1 Définir une structure **struct** noeud_s permettant de coder un nœud 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).

▶ **Correction**

```
typedef struct noeud_s {
    int valeur;
    struct noeud_s *gauche;
    struct noeud_s *droit;
} *noeud_t;

typedef noeud_t arbre_t;
```

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.

▶ **Correction**

```
#include <stdlib.h>

arbre_t cree_arbre(int valeur, arbre_t gauche, arbre_t droit) {
    arbre_t arbre = malloc(sizeof(struct noeud_s));
    arbre->valeur = valeur;
    arbre->gauche = gauche;
    arbre->droit = droit;
    return arbre;
}
```

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

▶ **Correction**

```
#include <stdlib.h>

void detruit_arbre(arbre_t arbre) {
    if (arbre == NULL)
        return;
    detruit_arbre(arbre->gauche);
    detruit_arbre(arbre->droit);
    free(arbre);
}
```

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

▶ **Correction**

```
int nombre_de_noeuds(arbre_t arbre) {
    if (arbre == NULL)
        return 0;
    return (1 + nombre_de_noeuds(arbre->gauche)
              + nombre_de_noeuds(arbre->droit));
}
```

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

▶ **Correction**

```
#include <stdio.h>

void affiche_arbre_rec(arbre_t arbre) {
    if (arbre != NULL) {
        affiche_arbre_rec(arbre->gauche);
        if (arbre->gauche != NULL)
            printf(",");
        printf("%d", arbre->valeur);
        if (arbre->droit != NULL)
            printf(",");
        affiche_arbre_rec(arbre->droit);
    }
}

void affiche_arbre(arbre_t arbre) {
    affiche_arbre_rec(arbre);
    printf("\n");
}
```

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

▶ **Correction**

```
#include <stdio.h>

void affiche_arbre2_rec(arbre_t arbre) {
    if (arbre == NULL)
        printf("_");
    else {
        printf("{");
        affiche_arbre2_rec(arbre->gauche);
        printf(",%d,", arbre->valeur);
        affiche_arbre2_rec(arbre->droit);
        printf("}");
    }
}

void affiche_arbre2(arbre_t arbre) {
```

```
    affiche_arbre2_rec(arbre);
    printf("\n");
}
```

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 nœuds se correspondant).

▶ Correction

```
int compare(arbre_t arbre1, arbre_t arbre2) {
    if (arbre1 == NULL)
        return (arbre2 != NULL);
    else { /* arbre1 != NULL */
        if (arbre2 == NULL)
            return 1;
        else /* arbre2 != NULL */
            /* on utilise l'évaluation paresseuse du || */
            return ((arbre1->valeur != arbre2->valeur)
                || compare(arbre1->gauche, arbre2->gauche)
                || compare(arbre1->droit, arbre2->droit));
    }
}
```

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

▶ Correction

```
arbre_t insere(arbre_t arbre, int valeur) {
    if (arbre == NULL)
        return cree_arbre(valeur, NULL, NULL);
    else {
        if (valeur < arbre->valeur)
            arbre->gauche = insere(arbre->gauche, valeur);
        else /* valeur >= arbre->valeur */
            arbre->droit = insere(arbre->droit, valeur);
        return arbre;
    }
}
```

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

▶ Correction

```
noeud_t trouve_noeud(arbre_t arbre, int valeur) {
    if (arbre == NULL)
        return NULL;
    if (valeur == arbre->valeur)
        return arbre;
    if (valeur < arbre->valeur) /* on descend à gauche */
        return trouve(arbre->gauche, valeur);
    else /* on descend à droite */
        return trouve(arbre->droite, valeur);
}
```

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 en argument...).

▶ Correction

```
/* à n'appeler que sur des arbres != NULL */
int verifie_rec(arbre_t arbre, int *min, int *max) {
    int i;
    *min = *max = arbre->valeur;
    if (arbre->gauche != NULL)
        /* on utilise l'évaluation paresseuse du || */
        if (!verifie_rec(arbre->gauche, &i, max) || !(arbre->valeur > *max))
            return 0;
    if (arbre->droit != NULL)
        /* on utilise l'évaluation paresseuse du || */
        if (!verifie_rec(arbre->droit, min, &i) || !(arbre->valeur <= *min))
            return 0;
    return 1;
}

int verifie(arbre_t arbre) {
    int min, max;
    return ((arbre == NULL) ? 1 : verifie_rec(arbre, &min, &max));
}
```

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...

▶ Correction

```
int tri_rec(arbre_t arbre, int i, int *tableau) {
    int j = 0;
    if (arbre == NULL)
        return j;
    j = tri_rec(arbre->gauche, i, tableau);
    tableau[i + j] = arbre->valeur;
    j++;
    j += tri_rec(arbre->droit, i + j, tableau);
    return j;
}

void tri(int taille, int *tableau) {
    int i;
    /* ne pas oublier d'initialiser à NULL l'arbre initial */
    arbre_t arbre = NULL;
    for (i = 0; i < taille; i++)
        arbre = insere(arbre, tableau[i]);
    tri_rec(arbre, 0, tableau);
    detruit_arbre(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 nœud contenant la valeur en question) :
- si le nœud à enlever ne possède aucun fils, on l'enlève,
- si le nœud 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.

▶ Correction

```
arbre_t supprime(arbre_t arbre, int valeur) {
    noeud_t noeud = arbre, *pere = &arbre;
    noeud_t nouveau_noeud, *nouveau_pere;
    while (noeud != NULL) {
        if (valeur == noeud->valeur)
            break;
        if (valeur < noeud->valeur) {
            pere = &noeud->gauche;
            noeud = noeud->gauche;
        } else { /* valeur >= noeud->valeur */
            pere = &noeud->droit;
            noeud = noeud->droit;
        }
    }
    if (noeud != NULL) {
        if (noeud->gauche == NULL) {
            if (noeud->droit == NULL) {
                *pere = NULL;
                free(noeud);
            } else { /* noeud->droit != NULL */
                *pere = noeud->droit;
                free(noeud);
            }
        } else { /* noeud->gauche != NULL */
            if (noeud->droit == NULL) {
                *pere = noeud->gauche;
                free(noeud);
            } else { /* noeud->droit != NULL */
                /* recherche de la plus petite valeur du sous-arbre droit */
                nouveau_noeud = noeud->droit;
                nouveau_pere = &noeud->droit;
                while (nouveau_noeud != NULL)
                    if (nouveau_noeud->gauche != NULL) {
                        nouveau_pere = &nouveau_noeud->gauche;
                        nouveau_noeud = nouveau_noeud->gauche;
                    }
                noeud->valeur = nouveau_noeud->valeur;
                *nouveau_pere = nouveau_noeud->droit;
                free(nouveau_noeud);
            }
        }
    }
    return arbre;
}
```