TP 8 : Arbres binaires de recherche

Semaine du 19 mars 2007

▶ **Exercice 1**

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

typedef noeud_t arbre_t;
```

▶ **Exercice 2**

```c
#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**

```c
#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**

```c
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**

```c
#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**

```c
#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**

```c
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**

```
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**

```
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**

```
/* 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**

```
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**

```
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;
}
```