

TD10 : Arbres

Programmation en C (LC4)

Semaine du 15 mai 2006

On travaille avec ce type d'arbres :

```
struct arbre {struct arbre * fils_gauche; struct arbre * fils_droit; int valeur};
```

1 Petits exercices

Exercice 1 Écrire une fonction calculant le nombre de nœuds d'un arbre.

► **Correction**

```
size_t compte_noeuds(struct arbre * a) {
    if (a)
        return 1+compte_noeuds(a->fils_gauche)+compte_noeuds(a->fils_droit);
    else
        return 0;
}
```

Exercice 2 Écrire une fonction calculant la profondeur d'un arbre.

► **Correction**

```
size_t profondeur(struct arbre * a) {
    if (a) {
        size_t hd=profondeur(a->fils_gauche);
        size_t hg=profondeur(a->fils_droit);
        return 1+(hd<hg?hg:hd);
    } else return 0;
}
```

Exercice 3 On dit qu'un arbre est « tournoi » si pour tout nœud, la valeur contenue dans ce nœud est inférieure à celles contenues dans ses fils. Écrire une fonction testant si un arbre est tournoi.

► **Correction**

```
int tournoi (struct arbre *a) {
    if (a->fils_gauche) {
        if (a->fils_droit) {
            return a->valeur<=a->fils_gauche->valeur&&a->valeur<=a->fils_droit->valeur&&
                tournoi(a->fils_gauche)&&tournoi(a->fils_droit);
        } else {
            return a->valeur<=a->fils_gauche->valeur&&tournoi(a->fils_gauche);
        }
    } else if (a->fils_droit) {
        return a->valeur<=a->fils_droit->valeur&&tournoi(a->fils_droit);
    } else return 1;
}
```

Exercice 4 On dit qu'un arbre est « listomorphe »¹ si le fils gauche de tous ses nœuds est vide. Écrire une fonction qui teste si un arbre est listomorphe.

► **Correction**

```
int listomorphe(struct arbre *a) {
    while (a) {
        if (a->fils_gauche) return 0;
        a=a->fils_droit;
    }
    return 1;
}
```

2 Parcours militaire

Le parcours militaire d'un arbre consiste à partir de la racine, puis parcourir les nœuds à profondeur 1 de gauche à droite, puis les nœuds à profondeur 2 (toujours de gauche à droite), puis les nœuds à profondeur 3, et ainsi de suite.

Exercice 5 Écrire une fonction `struct arbre ** parcours_militaire(struct arbre * a)`, qui effectue un parcours militaire de l'arbre `a`, et stocke, dans un tableau qu'elle alloue, les pointeurs vers les nœuds de l'arbre, dans l'ordre du parcours militaire, c'est-à-dire que la première case du tableau contient un pointeur vers la racine de l'arbre, la deuxième case contient un pointeur vers le fils gauche de la racine si il existe, et sinon vers le fils droit de la racine, la troisième case contient un pointeur vers le fils droit de la racine sauf si la racine n'avait pas de fils gauche, et ainsi de suite... Indice : dans le tableau où l'on construit le parcours, le dernier niveau parcouru correspond à un intervalle du tableau, il suffit de parcourir cette intervalle de gauche à droite, en enregistrant à la suite les fils gauche et droit des nœuds que l'on rencontre.

► **Correction**

```
struct arbre ** parcours_militaire(struct arbre * a) {
    size_t n=compte_noeuds(a);
    struct arbre ** res=malloc(n*sizeof(struct arbre *));
    struct arbre ** debut;
    struct arbre ** fin;
    res[0]=a;
    debut=res;
    fin=res+1;
    while (debut<fin) {
        struct arbre ** cur=fin;
        while (debut<fin) {
            if ((*debut)->fils_gauche) {
                *cur=(*debut)->fils_gauche;
                cur++;
            }
            if ((*debut)->fils_droit) {
                *cur=(*debut)->fils_droit;
                cur++;
            }
            debut++;
        }
        fin=cur;
    }
    return res;
}
```

3 Dessin d'arbre

On va voir un algorithme permettant de dessiner un arbre. On s'intéresse uniquement à l'algorithme de calcul des coordonnées des nœuds (en fait, seulement des abscisses), on ne se préoccupe pas de l'opération d'affichage.

¹ne ressortez pas ça dans une copie, c'est juste une définition ad-hoc pour cet exo

On va dessiner tous les nœuds d'un même niveau de l'arbre à la même ordonnée. La distance horizontale minimale entre deux nœuds sera une constante `DELTA_X` déclarée quelque part dans le programme. Chaque nœud sera centré horizontalement entre ses deux fils si il en a deux, et à `DELTA_X` à gauche (respectivement droite) de son fils droit (respectivement gauche) si il n'a pas de fils gauche (respectivement droit).

Le but sera de parcourir l'arbre en modifiant les champs `valeur` de sorte que le champ `valeur` d'un nœud contienne la différence entre son abscisse et celle de son père.

Une première méthode consiste à faire une fonction récursive qui dessine son argument, et renvoie en résultat la différence d'abscisse entre le nœud le plus à droite et le nœud le plus à gauche de l'arbre. L'idée est de faire un appel récursif sur chacun des deux fils, on obtient donc les largeurs l_1 et l_2 des fils gauche et droit. En plaçant le fils gauche (respectivement droit) à $(\Delta_x + l_1 + l_2)/2$ à gauche (respectivement droite) de la racine, on est sûr qu'à tous les niveaux, les fils gauche et droit seront séparés d'au moins Δ_x .

Exercice 6 Écrire une fonction récursive `int dessine_arbre_simple(struct arbre * a)` qui dessine un arbre par la méthode décrite ci-dessus.

En fait, cette méthode est très nettement sous-optimale, entre autres parce qu'on compare des abscisses de nœuds situés à des niveaux différents, alors que si les largeurs maximales ne sont pas atteintes aux mêmes niveaux, il peut y avoir moyen de rapprocher les fils gauche et droit. Pour y remédier, il suffit, au lieu de calculer la largeur de l'arbre, de calculer son « enveloppe », c'est-à-dire, pour chaque niveau de l'arbre, la différence d'abscisse entre son bord gauche (respectivement droit) et celui du niveau suivant. De la sorte, on peut parcourir en parallèle le bord droit du fils gauche et le bord gauche du fils droit, pour calculer la vraie distance minimale dont il faut écarter leurs racines.

Pour représenter ces enveloppes, le plus simple est d'utiliser deux listes chaînées, l'une pour le bord gauche, l'autre pour le bord droit. On se donne donc un type de liste chaînées, pour lequel on aura besoin des fonctions suivantes :

```
struct liste {int val;struct liste * suivant;};
struct liste * ajoute(int x,struct liste * l);
void concat(struct liste * a,struct liste * b);
void libere_intervalle(struct liste * a,struct liste * b);
```

La fonction `concat` avance jusqu'au bout de la liste `a`, et remplace le pointeur `NULL` final par `b`. Pour la fonction `libere_intervalle` `b` est censé pointer vers un maillon de la liste `a`. `libere_intervalle` parcourt la liste `a` jusqu'à rencontrer `b`, et appelle `free` sur tous les maillons rencontrés, sauf `b`.

Exercice 7 Implémenter les fonctions précédentes.

► Correction

```
struct liste * ajoute(int x,struct liste * l) {
    struct liste * r=malloc(sizeof(struct liste));
    r->val=x;
    r->suivant=l;
    return r;
}
struct liste * concat(struct liste * a,struct liste * b) {
    while (a) a=a->suivant;
    a->suivant=b;
}
struct liste * libere_intervalle(struct liste * a,struct liste * b) {
    while (a!=b) {
        struct liste * tmp;
        assert(a!=NULL);
        tmp=a->suivant;
        free(a);
        a=tmp;
    }
}
```

Exercice 8 Écrire une fonction

```
void dessine_arbre_optimal(struct arbre * a,
    struct liste ** bord_gauche,struct liste ** bord_droit)
```

qui dessine un arbre par l'algorithme décrit ci-dessus, en renvoyant dans `bord_gauche` la liste représentant l'enveloppe gauche de l'arbre, et dans `bord_droit` la liste représentant l'enveloppe droite.

► Correction

```

void dessine_arbre_optimal(struct arbre *a,
    struct liste ** bord_gauche, struct liste ** bord_droit) {
    struct liste * gg;
    struct liste * dg;
    struct liste * gd;
    struct liste * dd;
    if (a->fils_gauche) {
        if (a->fils_droit) {
            int xg=0; struct liste * bdg;
            int xd=0; struct liste * bgd;
            int dist=0;
            dessine_arbre_optimal(a->fils_gauche, &gg, &dg);
            dessine_arbre_optimal(a->fils_droit, &gd, &dd);
            bdg=dg; bgd=gd;
            while (bdg && bgd) {
                int tmp;
                xg+=bdg->val;
                xd+=bgd->val;
                tmp=xg-xd;
                if (tmp>dist) dist=tmp;
                bdg=bdg->suisvant;
                bgd=bgd->suisvant;
            }
            dist+=DELTA_X;
            dist/=2;
            a->fils_gauche->valeur=-dist;
            a->fils_droit->valeur=dist;
            *bord_gauche=ajoute(-dist, gg);
            *bord_droit=ajoute(dist, dd);
            if (bdg) {
                concat(dd, bdg);
                libere_intervalle(dg, bdg);
                libere_intervalle(gd, NULL);
            } else if (bgd) {
                concat(gg, bgd);
                libere_intervalle(gd, bgd);
                libere_intervalle(dg, NULL);
            }
        } else {
            dessine_arbre_optimal(a->fils_gauche, &gg, &dg);
            a->fils_gauche->valeur=-DELTA_X;
            *bord_gauche=ajoute(-DELTA_X, gg);
            *bord_droit=ajoute(-DELTA_X, dg);
        }
    } else if (a->fils_droit) {
        dessine_arbre_optimal(a->fils_droit, &gd, &dd);
        a->fils_droit->valeur=DELTA_X;
        *bord_gauche=ajoute(DELTA_X, gd);
        *bord_droit=ajoute(DELTA_X, dd);
    } else {
        *bord_gauche=NULL;
        *bord_droit=NULL;
    }
}

```