TELECOM Lille 1 **11 Avril 2013**

Nom : Prénom : N° : Durée : heures 2h

QROC

Module A31TTN

Structures de données

Aucun document n'est autorisé, les réponses sont à donner sur le présent document.

Exercice 1 – Récursivité (4 points)

1) Donner une fonction **<u>récursive</u>** en C pour le calcul de n^p, où n et p sont deux entiers naturels.

2) On peut faire un calcul plus rapide de n^p en partant de l'observation suivante :

$$n^p = \begin{cases} (n^{p/2}) * (n^{p/2}) & \text{si } p \text{ est pair} \\ (n^{p/2}) * (n^{p/2}) * n & \text{si } p \text{ est impair} \end{cases}$$

où $p/2$ représente la division entière de p par 2. Proposer une deuxième fonction qui calcule n^p.

Exercice 2 – Listes chaînées (8 points)

Soit P un polynôme de degré d et de coefficient p_i ($i \in [\![0, d]\!]$) : $P[X] = \sum_{i=0}^{d} p_i X^i$. On souhaite représenter P en occupant le moins d'éspace mémoire possible, en particulier quand le degré est élévé et/ou qu'il y a peu de coefficients non nuls. L'idée consiste à ne considérer que les monômes $p_i X^i$ « utile » c'est-à-dire tels que $p_i \neq 0$. Les monômes « utiles » seront stockés dans une liste chaînée dont chaque élément contiendra un **double** (le coefficient p_i du monôme) et un **int** (l'exposant i du monôme). Chaque coefficient d'un monôme « utile » sera ainsi chaîné via un pointeur au prochain monôme « utile » de degré strictement inférieur (ou NULL s'il n'y a plus de monôme « utile »).

On considère la structure de données suivante pour représenter cette liste chainée:

```
typedef struct mono {
        int degre;
        double coefficient;
        struct mono * suivant;
}monome;
typedef monome* polynome ;
```

La liste des fonctions et procédures données ci-dessous présenent des opérations sur les polynômes. On demandera de les expliquer ou de les compléter.

1- Expliquer ce que fait la fonction X suivante. Illustrer par un exemple.

```
void X(polynome p) {
    while (p != NULL) {
        if (p−>degre > 1) {
            printf("%f^%d", p−>coeffcient, p−>degre);
        } else {
            if (p−>degre == 1)
                printf("%f", p−>coeffcient);
            else
                printf("%f", p−>coeffcient);
            }
            p = p−>suivant;
            if (p != NULL)
                printf("+");
    }
    printf("\n");
}
```

2- La fonction ***cree_monome*** suivante alloue et initialise la mémoire nécessaire à un monôme. Compléter cette fonction :

```
monome* cree_monome(int degre, double coeffcient) {
      monome* m = malloc(sizeof(_____));
      _____;
      _____;
      _____;
      return m;
}
```

3- La procédure ***detruit_polynome*** qui libère toute la mémoire utilisée par un polynôme. On demande aussi de la compléter.

```
void detruit_polynome(polynome p) {
      monome m;
      while (p != NULL) {
      _____;
      _____;
      _____;
      }
}
```

4- Expliquer en illustrant par un exemple ce que fait la fonction ***Y*** suivante.

```
polynome Y(polynome p)
{
       polynome q = NULL;
       monome* m;
       while (p != NULL && p−>degre != 0) {
             m = cree_monome(p−>degre − 1, p −>degre *p −>coeffcient);
             if(q==NULL)
                   q=m;
              else
                    m = q−>suivant;
              p = p−>suivant;
       }
   return q;
}
```

5- Exercice 3 – Les arbres binaires (8 points)

On considère la suite de nombres suivante : 1, 3, -50, -32, -38, -46, -67, -66, -69, -12, 1, -70.

1) Partant d'un arbre binaire vide, on ajoute successivement les valeurs de la suite ci-dessus. Donnez l'arbre binaire résultat *(il ne s'agit pas de donner la version équilibrée de l'arbre)*.

2) Vérifier si l'arbre obtenu est équilibré, s'il n'est pas équilibré, équilibrez-le. Expliquez les opérations de rotation effectuées *(on recommencera l'insertion dans un arbre vide)*.

3) On considère la structure de données suivante pour représenter cet arbre :

```
typedef struct Maillon {
        int Noeud;
        struct Maillon * SAG;
        struct Maillon * SAD;
} Node;
Typedef Node* Arbre ;
```

On considère aussi la procédure suivante ***Mystere***:

```
void Mystere (Arbre Racine) {
        if (Racine)
        {
                printf("%d ",Racine->Noeud);
                Mystere (Racine->SAG);
                Mystere (Racine->SAD);
        }
}
```

a) Expliquez ce que fait cette fonction ***Mystere***? Donnez alors un non significatif à cette fonction.

b) Donnez le résultat de cette fonction sur l'arbre originale *(non équilibré)* obtenu dans 1).

c) Donner aussi le résultat sur l'arbre équilibré obtenu dans 2).

4) Proposez une fonction récursive qui permet de calculer le produit des feuilles.

5) Proposez une fonction récursive qui permet de retourner la plus petite valeur.

Bonne chance.