# TD N° 2
# Solutions des exercices 6 et 7

## Exercice 6

On considère des polynômes de la forme :

$$P(X) = a_nX^{e1} + a_{n-1}X^{e2} + \ldots + a_1X^{en}$$

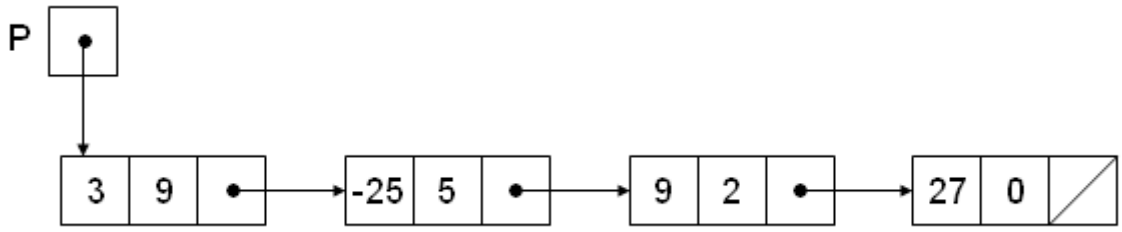
Avec $e_1 > e_2 > \ldots > e_n \geq 0$ des entiers et $a_n, a_{n-1}, \ldots, a_1$ des réels non nuls.

1. Trouver une représentation adéquate d'un polynôme par une liste chaînée.
2. Écrire des procédures qui effectuent les opérations suivantes : (i) le calcul de la valeur d'un polynôme P pour un réel X donné, (ii) l'affichage d'un polynôme, (iii) l'addition de deux polynômes, (iv) la soustraction de deux polynômes, (v) la multiplication de deux polynômes, et (vi) la dérivée d'un polynôme.

**Solutions :**

1. Il est naturel de représenter un polynôme $P(X) = a_nX^{e1} + a_{n-1}X^{e2} + \ldots + a_1X^{en}$ par une liste simplement chaînée de monômes $P = (M_n, M_{n-1}, \ldots, M_1)$, ordonnée par degré décroissant, où chaque monôme $M_k = (a_k, e_{n-k+1})$.

Par exemple, le polynôme $P(x) = 3X^9 - 25X^5 + 9X^2 + 27$ sera représenté par la liste suivante :




Un nœud de cette liste est une structure composée de trois champs : un réel (coefficient du monôme), un entier (degré du monôme) et un pointeur vers un autre nœud. On peut alors représenter ce nœud comme suit :

```
typedef struct Noeud
{
  float  coef;          /* coefficient du monôme */
  int    deg;           /* degré du monôme */
  struct Noeud* suiv;   /* pointeur vers le monôme suivant */
} Noeud;
```

Le type polynôme sera défini comme étant un pointeur vers le premier monôme de plus haut degré :

```
typedef Noeud* polynome;
```

Comme **opérations de base** pour ce type, on peut donner :

- ✓ Noeud* obtenirNoeud(float c, int d) : crée un nœud (un monôme) en mémoire à partir de son coefficient c et son degré d. Elle retourne un pointeur vers ce monôme.
- ✓ void creer(polynome* P, int n) : crée un polynôme ayant n monômes et qui sera pointé par P.
- ✓ int estNul(polynome P) : teste si un polynôme P est identiquement nul (P(X) = 0, pour tout X). Il serait avantageux de représenter le polynôme Nul par une liste vide.
- ✓ int estConstant(polynome P) : teste si un polynôme P est constante (P(X) = C ≠ 0, pour tout X).

**Code C de ces opérations** :

```
/*** La fonction « obtenirNoeud » ***/

Noeud* obtenirNoeud(float c, int d)
{
  Noeud* temp;

  temp = (Noeud*) malloc(sizeof(Noeud));
  temp->coef = c;
  temp->deg = d;
  temp->suiv = NULL;
  return temp;
}

  /*** La procédure « creer » crée un polynôme ayant n monômes qui seront
              saisies au clavier (n >= 1), par insertion à la fin ***/

void creer(polynome* P, int n)
{
  int i, d;
  float c;
  Noeud *temp;

  /* création du premier nœud de la liste */

  printf("coefficient[%d] = ", n);
  scanf("%f", &c);
  printf("degre[%d]       = ", n);
  scanf("%d", &d);
  temp = obtenirNoeud(c, d);
  *P = temp;   /* chaînage */

  /* création des (n -1) nœuds restants par insertion à la fin */
```

```
  for(i = n - 1; i > 0; i--)
  {
    printf("coefficient[%d] = ", i);
    scanf("%f", &c);
    printf("degre[%d]      = ", i);
    scanf("%d", &d);
    temp->suiv = obtenirNoeud(c, d);
    temp = temp->suiv;
  }
}
```

/*** La fonction « estNul » ***/

```
int estNul(polynome P)
{
  return P == NULL;
}
```

/*** La fonction « estConstant » ***/

```
int estConstant(polynome P)
{
  return    (!estNul(P)) && P->deg == 0;
}
```

2. Implémentation des opérations mathématiques sur les polynômes :

/*** La fonction « evaluer » calcule la valeur de P(X) ***/

```
float evaluer(polynome P, float x)
{
  Noeud* temp = P;
  float res = 0.0;

  while(temp != NULL)
  {
    res += temp->coef * pow(x, temp->deg);
    temp = temp->suiv;
  }
  return res;
}
```

/*** La procédure « afficher » ***/

```
void afficher(polynome P)
{
  if(estNull(P))
    printf("Polynome nul : P(X) = 0\n");
```

```
  else
   {
     Noeud* temp = P;

     while(temp->suiv != NULL)
     {
       printf("%.2fX^%d + ", temp->coef, temp->deg);
       temp = temp->next;
     }
     printf("%.2fX^%d\n", temp->coef, temp->deg);
  }
}

 /*** La fonction « somme » calcule la somme de deux polynômes P et Q ***/

polynome  somme(polynome P, polynome Q)
{
  if(estNul(P))
    return Q;
  if(esNul(Q))
    return P;
  else
  {
     polynome R;

     if(P->deg == Q->deg)
     {
        float c = P->coef + Q->coef;

        if(c == 0)
          R = somme(P->suiv, Q->suiv);
        else
        {
          R = obtenirNoeud(c, P->deg);
          R->suiv = somme(P->suiv, Q->suiv);
        }
     }
     else
     {
       if(P->deg > Q->deg)
       {
          R = obtenirNoeud(P->coef, P->deg);
          R->suiv = somme(P->suiv, Q);
       }
       else
       {
          R = obtenirNoeud(Q->coef, Q->deg);
          R->suiv = somme(P, Q->suiv);
        }
```

```
    }
    return R;
}

      /*** La fonction « oppose » retourne l'opposé d'un polynôme P ***/

polynome oppose(polynome P)
{
  polynome R = NULL;

  if(!estNul(P))
  {
    Noeud* temp1;
    Noeud* temp2;

    R = obtenirNoeud(-P->coef, P->deg);
    temp1 = R;
    temp2 = P->suiv;
    while(temp2 != NULL)
    {
      temp1->suiv = obtenirNoeud(-temp2->coef, temp2->deg);
      temp1 = temp1->suiv;
      temp2 = temp2->suiv;
    }
    return R;
  }
}

 /*** La fonction « diff »retourne la différence de deux polynômes P et Q ***/

polynome  diff(polynome P, polynome Q)
{
  return somme(P, oppose(Q));
}

/*** La fonction « prodPolyMonome » retourne le produit d'un polynôme P par
                                un monôme M ***/

polynome  prodPolyMonome(polynome P, Noeud* M)
{
  polynome R = NULL;
  float c = M->coef;
  int d = M->deg;

  if(!estNul(P))
  {
    Noeud* temp1;
    Noeud* temp2;
```

```
    R = obtenirNoeud(c * P->coef, P->deg + d);
    temp1 = R;
    temp2 = P->suiv;
    while(temp2 != NULL)
    {
      temp1->suiv = obtenirNoeud(c * temp2->coef, temp2->deg + d);
      temp1 = temp1->suiv;
      temp2 = temp2->suiv;
    }
    return R;
  }
}

 /*** La fonction « produit » calcule le produit de deux polynômes P et Q ***/

polynome produit(polynome P, polynome Q)
{
  polynome R = NULL;
  Noeud* temp = Q;

  while(temp != NULL)
  {
    R = somme(R, prodPolyMonome(P, temp));
    temp = temp->suiv;
  }
  return R;
}

      /*** La fonction « derivee » calcule la dérivée d'un polynôme P ***/

polynome derivee(polynome P)
{
  if(esNul(P) || estConstant(P))
    return NULL;
  else
  {
    polynome R;

    R = obtenirNoeud(P->coef * P->deg, P->deg - 1);
    R->suiv = derivee(P->suiv);
    return R;
  }
}

      /*** Un programme principal pour tester les diverses fonctions ***/

main( )
{
  polynome P, Q, R;
```

```
  int n, m;
  float X;

  printf("\nNombre de monomes du polynome P : ");
  scanf("%d", &n);
  creer(&P, n);
  printf("\tP(X) = ");
  afficher(P);

  printf("\nNombre de monomes du polynome Q : ");
  scanf("%d", &m);
  creer(&Q, m);
  printf("\tQ = ");
  afficher(Q);

  printf("Donner la valeur de X : ");
  scanf("%f", &X);
  printf(P(X) = %.2f, Q(X) = %.2f\n", evaluer(P, X), evaluer(Q, X));

  printf("\nSomme : P + Q = ");
  R = somme(P, Q);
  afficher(R);

  printf("\nDifference : P - Q = ");
  R = diff(P, Q);
  afficher(R);

  printf("\nProduit : P * Q = ");
  R = produit(P, Q);
  afficher(R);

  printf("\nDerivee : P'(X) = ");
  R = derivee(P);
  afficher(R);

  printf("\nDerivee : Q'(X) = ");
  R = derivee(Q);
  afficher(R);

  system("pause");
}
```

## Exercice 7

Écrire les procédures qui décrivent les opérations d'insertion et de suppression pour :

1. les listes doublement chaînées.
2. les listes circulaires.
3. les listes circulaires doublement chaînées.

**Solutions :**

1. Les listes doublement chaînées :

**Définition du type** :

```
typedef struct Noeud
{
  struct Noeud* prec;  /* pointeur vers le nœud précédent */
  int  info;  /* contenu */
  struct Noeud* suiv;  /* pointeur vers le nœud suivant */
} Noeud;

typedef Noeud* listeDouble;
```

**Opérations de base** :

```
                /*** Fonction de création d'un nœud ***/

Noeud* obtenirNoeud(int x)
{
  Noeud* nouv;   /* une variable désignant un nouveau nœud */

  nouv = (Noeud*) malloc(sizeof(Noeud));
  nouv->info = x;
  nouv->suiv = NULL;
  nouv->prec = NULL;
  return nouv;
}
          /*** Insertion d'un élément au début ***/

void insererDebut(listeDouble* pdL, int x)
{
  Noeud* nouv;

  nouv = obtenirNoeud(x);
  if(*pdL != NULL)
  {
    (*pdL)->prec = nouv;
    nouv->suiv = *pdL;
  }
  *pdL = nouv;
}

           /*** Insertion d'un élément à la fin ***/

void insererFin(listeDouble* pdL, int x)
{
  Noeud* nouv;
```

```
  nouv = obtenirNoeud(x);
  if(*pdL == NULL)
    *pdL = nouv;
  else
  {
    Noeud* temp = *pdL;

    while(temp->suiv != NULL)
          temp = temp->suiv;
    nouv->prec = temp;
    temp->suiv = nouv;
  }
}

                /*** Suppression au début ***/

void supprimerDebut(listeDouble* pdL)
{
  if(*pdL != NULL)
  {
    Noeud* temp = *pdL;

    *pdL = (*pdL)->suiv;
    if(*pdL != NULL)
      (*pdL)->prec = NULL;
    free(temp);
  }
}

                /*** Suppression à la fin ***/

void supprimerFin(listeDouble* pdL)
{
  if(*pdL != NULL)
  {
    Noeud* temp = *pdL;
    Noeud* dernier;

    while(temp->suiv != NULL)
          temp = temp->suiv;
    if(temp == *pdL)
    {
      free(temp);
      *pdL = NULL;
    }
    else
    {
       dernier = temp;
```

```
            temp->prec->suiv = NULL;
            free(dernier);
        }
    }
}
                /*** Affichage d'une liste double ***/

void afficher(listeDouble L)
{
  Noeud* temp = L;

  if(L == NULL)
    puts("liste vide");
  else
  {
    while(temp != NULL)
    {
      printf("%d ", temp->info);
      temp = temp->suiv;
    }
    printf("\n");
  }
}
```

/*** Recherche d'un élément : cette fonction retourne un pointeur vers le nœud qui contient la première occurrence de l'élément x dans une liste double L. Si x n'existe pas, elle retourne le pointeur **NULL** ***/

```
Noeud* chercher(listeDouble L, int x)
{
  Noeud* temp = L;

  while(temp != NULL)
  {
    if(temp->info == x)
      return temp;
    temp = temp->suiv;
  }
  return NULL;
}
```

2. Les listes circulaires :

**Définition du type** :

```
typedef struct Noeud
{
  int  info;  /* contenu */
  struct Noeud* suiv;  /* pointeur vers le nœud suivant */
} Noeud;
```

```
typedef Noeud* listeCirculaire;
```

**Opérations de base** :

```
                /*** Fonction de création d'un nœud ***/

Noeud* obtenirNoeud(int x)
{
  Noeud* nouv;   /* une variable désignant un nouveau nœud */

  nouv = (Noeud*) malloc(sizeof(Noeud));
  nouv->info = x;
  nouv->suiv = NULL;
  return nouv;
}
          /*** Insertion d'un élément au début ***/

void insererDebut(listeCirculaire* pCL, int x)
{
  Noeud* nouv;
  Noeud* temp;

  nouv = obtenirNoeud(x);
  if(*pCL == NULL)
  {
    *pCL = nouv;
    nouv->suiv = *pCL;
  }
  else
  {
    temp = *pCL;
    while(temp->suiv != *pCL)
          temp = temp->suiv;
    nouv->suiv = *pCL;
    *pCL = nouv;
    temp->suiv = *pCL;
  }
}
           /*** Insertion d'un élément à la fin ***/

void insererFin(listeCirculaire* pCL, int x)
{
  Noeud* nouv = obtenirNoeud(x);
  Noeud* temp;

  if(*pCL == NULL)
  {
    *pCL = nouv;
```

```
    nouv->suiv = *pCL;
  }
  else
  {
    temp = *pCL;
    while(temp->suiv != *pCL)
        temp = temp->suiv;
    temp->suiv = nouv;
    nouv->suiv = *pCL;
  }
}
                /*** Suppression au début ***/

void supprimerDebut(listeCirculaire* pCL)
{
  if(*pCL != NULL)
  {
    if((*pCL)->suiv == *pCL)
    {
      free(*pCL);
      *pCL = NULL;
    }
    else
    {
      Noeud* prem;
      Noeud* dern;

      prem = dern = *pCL;
      while(dern->suiv != *pCL)
          dern = dern->suiv;
      *pCL = (*pCL)->suiv;
      dern->suiv = *pCL;
      free(prem);
    }
  }
}
                /*** Suppression à la fin ***/

void supprimerFin(listeCirculaire *pCL)
{
  if(*pCL != NULL)
  {
    if((*pCL)->suiv == *pCL)
    {
      free(*pCL);
      *pCL = NULL;
    }
    else
    {
```

```
        Noeud* dern;
        Noeud* avant;

        avant = dern = *pCL;
        while(dern->suiv != *pCL)
        {
          avant = dern;
          dern = dern->suiv;
        }
        avant->suiv = *pCL;
        free(dern);
      }
    }
}

/*** Recherche d'un élément x dans une liste circulaire L ***/

Noeud* chercher(listeCirculaire L, int x)
{
  Noeud* temp = L;

  if(temp != NULL)
  {
    do
     {
       if(temp->info == x)
         return temp;
       temp = temp->suiv;
     } while(temp != L);
  }
  return NULL;
}

        /*** Affichage d'une liste simple circulaire ***/

void afficher(listeCirculaire L)
{
  if(L != NULL)
  {
    Noeud* temp = L;

    do
    {
      printf("%d ", temp->info);
      temp = temp->suiv;
    } while(temp != L);
    printf("\n");
  }
  else
```

```
        puts("liste vide");
    }
```

3. Les listes circulaires doublement chaînées

**Définition du type** :

```
typedef struct Noeud
{
  struct Noeud* prec;  /* pointeur vers le nœud précédent */
  int  info;           /* contenu */
  struct Noeud* suiv;  /* pointeur vers le nœud suivant */
} Noeud;

typedef Noeud* listeDbleCirc;
```

**Opérations de base** :

```
                /*** Fonction de création d'un nœud ***/

Noeud* obtenirNoeud(int x)
{
  Noeud* nouv;

  nouv = (Noeud*) malloc(sizeof(Noeud));
  nouv->info = x;
  nouv->suiv = NULL;
  nouv->prec = NULL;
  return nouv;
}
          /*** Insertion d'un élément au début ***/

void insererDebut(listeDbleCirc* pL, int x)
{
  Noeud* nouv;
  Noeud* temp;

  nouv = obtenirNoeud(x);
  if(*pL == NULL)
  {
    *pL = nouv;
    nouv->suiv = *pL;
    nouv->prec = *pL;
  }
  else
  {
    nouv->prec = (*pL)->prec;
    nouv->suiv = *pL;
    (*pL)->prec->suiv = nouv;
    (*pL)->prec = nouv;
    *pL = nouv;
```

```
  }
}

        /*** Insertion d'un élément à la fin ***/

void insererFin(listeDbleCirc* pL, int x)
{
  Noeud *nouv, *temp;

  nouv = obtenirNoeud(x);
  if(*pL == NULL)
  {
    *pL = nouv;
    nouv->suiv = *pL;
    nouv->prec = *pL;
  }
  else
  {
    nouv->prec = (*pL)->prec;
    nouv->suiv = *pL;
    (*pL)->prec->suiv = nouv;
    (*pL)->prec = nouv;
  }
}

void supprimerDebut(listeDbleCirc* pL)
{
  Noeud* temp;

  if(*pL != NULL)
  {
    if((*pL)->suiv == *pL)
    {
      free(*pL);
      *pL = NULL;
    }
    else
    {
      temp = *pL;
      *pL = (*pL)->suiv;
      temp->prec->suiv = *pL;
      (*pL)->prec = temp->prec;
      free(temp);
    }
  }
}

void supprimerFin(listeDbleCirc *pL)
{
```

```
  Noeud *temp;

  if(*pL != NULL)
  {
    if((*pL)->suiv == *pL)
    {
      free(*pL);
      *pL = NULL;
    }
    else
    {
      temp = *pL;
      while(temp->suiv != *pL)
          temp = temp->suiv;
      temp->prec->suiv = temp->suiv;
      temp->suiv->prec = temp->prec;
      free(temp);
    }
  }
}

void afficher(listeDbleCirc L)
{
  if(L != NULL)
  {
    Noeud* temp = L;
    do
    {
      printf("%d ", temp->info);
      temp = temp->suiv;
    }while(temp != L);
    printf("\n");
  }
  else
   puts("liste vide");
}

Noeud* chercher(listeDbleCirc L, int x)
{
  Noeud* temp = L;

  if(temp != NULL)
  {
    do
     {
       if(temp->info == x)
         return temp;
       temp = temp->suiv;
     } while(temp != L);
```

```
    }
    return NULL;
}
```

## Exercice 2

On se donne N entiers en lecture. Écrire une procédure qui constitue une liste chaînée de ces entiers. On donnera une version itérative et une version récursive. On pourra utiliser la fonction « creerCellule » pour créer un élément (voir exercice 1).

**Solution récursive**

Voici une procédure récursive qui crée une liste simplement chaînée par insertion des N entiers saisis au clavier au début de la liste.

```
typedef struct cellule
{
  int info;
  struct cellule *suiv;
} cellule;

typedef cellule *Liste;

void insertBeginRec(Liste *pL, int N)
{
  if(N != 0)
  {
     cellule *nouveau;
     int x;

     printf("donner x : ");
     scanf("%d", &x);
     nouveau = creerCellule(x);
     if(*pL == NULL)
       *pL = nouveau;
     else
     {
        nouveau->suiv = *pL;
        *pL = nouveau;
     }
     insertBeginRec(pL, N – 1);
  }
}
```

Voici une procédure récursive qui crée une liste simplement chaînée par insertion des N entiers saisis au clavier à la fin de la liste :

```
void insertEndRec(Liste *pL, int N)
{
  if(N != 0)
  {
```

```
        cellule *nouveau;
        int x;

        printf("donner x : ");
        scanf("%d", &x);
        nouveau = creerCellule(x);
        *pL = nouveau;
        insertEndRec(&(*pL)->suiv, N – 1);
    }
}
```