SOLUTION TD LES POINTEURS ET LES LISTES !

Exercice No 1

Soit une matrice (N,M) d'entiers. Ecrire un algorithme qui génère deux listes à partir de cette matrice.
1- La première regroupe les maximums des lignes (FIFO);
2- Et, la deuxième la somme des colonnes (LIFO).

Solution

```
Algorithme  ListeMatrice;
Type
  pList = ^listElt;
  listElt = Enregistrement
      val : Entier;
      svt : pList;
  Fin;
  Matrice = Tableau[1..50, 1..30] de Entier;
Var
  LLig, LCol: pListe;
  A : Matrice;
  i, J, N, M : Entier;

Fonction    MaxLigne(A:Matrice, M:Entier, i:Entier):Entier
Var
   max, j : Entier;
Debut
   max <-- A[i, 1];
   Pour J <-- 2 a M Faire
      Si (A[i, J] > max) alors
         max <-- A[i, J];
      Fsi
   Fait
   Retourner max;
Fin;

Fonction    SomColonne(A:Matrice, N:Entier, J:Entier):Entier
Var
   som, i : Entier;
Debut
   som <-- 0;
   Pour i <-- 1 a N Faire
      Som <-- Som + A[i, J];
   Fait
   Retourner Som;
```

```
Fin;

Procedure    AfficherListe(E/ txt: Chaine, E/ L : pList)
Debut
   Ecrire(txt);
  Tant Que (L <> Nil) Faire
     Ecrire(L^.val);
     L <--- L^.svt;
  Fait
Fin;

// creation Liste LiFo
Fonction    CreerListeLifo(A: Matrice, N, M:Entier):pList
Var
  L, p : pList;
  J:Entier;
Debut
  L <-- Nil;
  Pour J <-- 1 a M Faire
      Allouer(p);
      p^.val <--- SomColonne(A, N, J);
      p^.svt <-- L;
      L <-- p;
  Fait

  Retourner L;
Fin;

// creation Liste FiFo
Fonction     CreerListeFifo(A: Matrice, N, M:Entier):pList
Var
  L, p, prcd : pList;
  i:Entier;
Debut
  Allouer(p);
  L <--- p;
  p^.val <-- MaxLigne(A, M, 1);
  prcd <-- p;
  Pour i <-- 2 a N Faire
      Allouer(p);
      p^.val <--- MaxLigne(A, M, i);
      prcd^.svt <--- p;
      prcd <-- p;
  Fait
```

```
  p^.svt <-- Nil;

  Retourner L;
Fin;

Fonction    LibererListe(L : ^listElt):pList
Var
  p: pList;
Debut
  Tant Que (L <> Nil) Faire
     p <-- L;
     L <-- L^.svt;
     Liberer(p);
  Fait
  Retourner Nil;
Fin;

Debut
  Repeter
    Lire(N);
  Jusqu'a ((N >= 1) et (N <= 50));
  Repeter
    Lire(M);
  Jusqu'a ((M >= 1) et (M <= 30));
  Pour i <-- 1 a N  Faire
      Pour J <-- 1 a M  Faire
         Lire(A[i, J]);
      Fait
  Fait

  LLig <--- CreerListeFifo(A, N, M);
  LCol <--- CreerListeLifo(A, N, M);
  AfficherListe("Liste FIFO Max des Lignes", LLig);
  AfficherListe("Liste LIFO Somme des Colonne", LCol);

  LibererListe(LLig);
  LibererListe(LCol);
Fin.
```

EXERCICE 2 :

Soit une liste d'entiers L, écrire les actions paramétrées suivantes permettant :

1- La suppression des doublons (éléments identiques) ;

2- La suppression de la valeur minimale d'une liste;
3- La création de la liste miroir de L (avec ensuite sans création d'une nouvelle liste);
4- La fusion de deux listes triées d'entiers L1 et L2 en une liste triée L3 (avec ensuite sans création d'une nouvelle liste);

Solution

```
Type
  pList = ^listElt;
  listElt = Enregistrement
      val : Entier;
      svt : pList;
  Fin;

2-1 La suppression des doublons (éléments identiques)
Procedure      supressionDoublons(E/ L: pList)
Var
     ptr1, ptr2, prcd : pList;
Debut
   Si (L <> Nil) alors
         ptr1 <--- L;
       /* Traverser la liste element par element */
       Tant Que (ptr1^.svt <> Nil)
       Faire
            ptr2 <--- ptr1^.svt;
            prcd <--- ptr1;
          /* Comparer l'element courant ptr1 avec le reste des elements
*/
          Tant Que (ptr2 <> Nil)
          Faire
                  /* Si doublon alors supprimer le doublon */
                  Si (ptr1^.val = ptr2^.val) alors
                          /* Mettre dans le svt du precedent le svt
                             de l'element a liberer */
                        prcd^.svt <--- ptr2^.svt;
                        Liberer(ptr2);
                        ptr2 <--- prcd^.svt;
                  Sinon
                        prcd <--- ptr2;
                        ptr2 <--- ptr2^.svt;
                 Fsi
          Fait
          ptr1 <--- ptr1^.svt;
```

```
        Fait
     Fsi
Fin;
```

2-2 La suppression de la valeur minimale d'une liste

```
Fonction    SupprimerMin(E\ L: pList) : pList
Var
   p, prcd, pMin, pMinPrcd : pList;
Debut
   Si (L <> Nil)  alors
      pMin <--- L; /* supposer le Min c'est le 1er */
      p <--- L^.svt;  /* commencer les comparaison a partir du 2eme */
      prcd <-- L;  /* le precedent du 2eme element est le 1er element */
      Tant Que (p <> Nil)  Faire   /* parcourir tous les elements de la
liste pour la comparaison */
         Si (p^.val < pMin^.val) alors
            pMin <--- p;
            pMinPrcd <--- prcd;
         Fsi
         prcd <--- p;
         p <--- p^.svt;
      Fait
      Si (pMin = L) alors /* Le minimum est le premier */
        L <--- L^.svt;
      Sinon
        pMinPrcd^.svt <--- pMin^.svt;
      Fsi
      Liberer(pMin);
   Fsi
   Retourner   L;
Fin;
```

2-3-1 La création de la liste miroir de L

```
Fonction       creationListeMiroirAvecNouvelleListe(L: pList) : pList
Var
    p, L_miroir : pList;
Debut
    L_miroir <--- Nil;

    /* traverser les elements de la liste 1 et ajouter chaque element
       en tete de la liste 2  de cette facon on obtient une liste inversee:
       creation d'une liste LIFO (liste L_miroir) a partir d'une premiere
liste (liste L).
     */
```

```
    Tant Que (L <> Nil)
    Faire
        Allouer(p);
        p^.val <--- L^.val;
        p^.svt <--- L_miroir;
        L_miroir <--- p;
        L <-- L^.svt;
    Fait

    retourner L_miroir;
Fin;
```

2-3-2 Liste Miroir sans creation de nouvelle liste

```
Fonction      creationListeMiroirSansNouvelleListe(L: pList) : pList
Var
   p, prcd, pSuiv : pList;
Debut
   /* initialiser prcd a NiL. mettre dans p l'adresse du 1er element de
la liste
       Chaque element va avoir son suivant le precedent de la liste
originale,
       l'ancien 1er aura comme suivant NiL.
    */
    p <--- L;
    prcd <--- Nil;
    Tant Que (p <> Nil)
    Faire
        /* le courant est p, sauvegarder dans pSuiv l'adresse du
suivant */
        pSuiv <--- p^.svt;
        p^.svt <--- prcd;     /* le suivant de p devient le precedent */
        prcd <--- p;           /* a la prochaine iteration le courant devient
le precedent */
        p <--- pSuiv;          /* et le suivant devient le courant */
    Fait

    retourner   prcd;
Fin;
```

2.4.1 La fusion de deux listes triées d’entiers L1 et L2 en une liste triée L3 avec création d’une nouvelle liste;

On utilisera dans cet exercice les 2 fonctions ajoutEnTete() et ajoutApres().

```
/* Ajoute un element en tete de liste */
Fonction     ajoutEnTete(L: pList, V: Entier) : pList
Var
   p : pList;
Debut
   Allouer(p);
   p^.val <--- V;
   p^.svt <--- L;

   Retourner p;
Fin;

/* ajoute d'un élément  après une adresse donnée prcd */
Fonction     ajoutApres(prcd: pList, V: Entier) : pList
Var
    p: pList;
Debut
    Allouer(p);
    p^.val <--- V;
    p^.svt <--- prcd^.svt;
    prcd^.svt <--- p;

    retourner p;
Fin;

Fonction  FusionL1L2AvecAlloc(L1, L2: pList) : pList
Var
   L, L12, prec: pList;
   val : Entier;
Debut
   L <--- Nil;
   Si ((L1 <> Nil) et (L2 <> Nil)) alors
        /* Mettre le lien du plus petit dans L */
        Si (L1^.val < L2^.val) alors
            val <--- L1^.val;
            L1 <--- L1^.svt;
        Sinon
            val <--- L2^.val;
            L2 <--- L2^.svt;
         Fsi
         L <--- AjoutEnTete(L, val);
         prec <--- L;
         Tant Que ((L1 <> Nil) et (L2 <> Nil))
```

```
        Faire
              Si (L1^.val < L2^.val) alors
                    val <--- L1^.val;
                    L1 <--- L1^.svt;
              Sinon
                    val <--- L2^.val;
                    L2 <--- L2^.svt;
               Fsi
              prec <--- AjoutApres(prec, val);
        Fait
        /* Ajouter les liens qui manquent de L1 ou L2 */
       Tant Que (L1 <> Nil)
        Faire
            val <--- L1^.val;
            L1 <--- L1^.svt;
            prec <--- AjoutApres(prec, val);
        Fait
        Tant Que (L2 <> Nil)
        Faire
              val <--- L2^.val;
               L2 <--- L2^.svt;
              prec <--- AjoutApres(prec, val);
         Fait
Sinon
        Si ((L1 <> Nil) OU (L2 <> Nil))
            Si (L1 <> Nil) alors
                  L12 <-- L1;
            Sinon
                   L12 <-- L2;
            Fsi
            /* Allouer le 1er element */
             val <--- L12^.val;
              L <--- AjoutEnTete(L, val);
              prec <--- L;
              L12 <--- L12^.svt;
             Tant Que (L12 <> Nil)
             Faire
                  val <--- L12^.val;
                  L12 <--- L12^.svt;
                  prec <--- AjoutApres(prec, val);
             Fait
        Fsi
Fsi
```

```
    Retourner L;
Fin;
```

2.4.2 La fusion de deux listes triées d'entiers L1 et L2 en une liste triée L3 sans création d'une nouvelle liste;

```
Fonction      FusionL1L2SansAlloc(L1, L2: pList) : pList
Var
  L, prec: pList;
Debut
  Si ((L1 <> Nil) et (L2 <> Nil)) alors
     /* Mettre le lien du plus petit dans L */
     Si (L1^.val < L2^.val) alors
        L <--- L1;
        L1 <--- L1^.svt;
     Sinon
        L <--- L2;
        L2 <--- L2^.svt;
     Fsi
     prec <--- L;
     Tant Que ((L1 <> Nil) et (L2 <> Nil))
     Faire
        Si (L1^.val < L2^.val) alors
           prec^.svt <--- L1;
           L1 <--- L1^.svt;
        Sinon
           prec^.svt <--- L2;
           L2 <--- L2^.svt;
        Fsi
        prec <--- prec^.svt;
     Fait
     /* Ajouter les liens qui manquent de L1 ou L2 */
     Si (L1 <> Nil)  alors
        prec^.svt <--- L1;
     Sinon
        prec^.svt <--- L2;
     Fsi
  Sinon
    Si (L1 <> Nil) alors
       L <--- L1;
    Sinon
       L <--- L2;
    Fsi
  Fsi
```

```
   Retourner L;
Fin;
```

EXERCICE 3 :

Soit L une liste d'entiers. Ecrire une procédure qui permet d'éclater la liste L en deux listes : Lp contenant les entiers positifs et Ln contenant les entiers négatifs. (Sans création de nouvelles listes).

Solution

On vas donc traverser la liste L element par element et on vas ajouter l'element a la fin de la liste (Lp) sans allocation de nouveau element si la valeur de l'element est >= 0 ou bien on vas ajouter l'element a la fin de la liste (Ln) sans allocation de nouveau element si la valeur de l'element est < 0. On ne vas pas ecrire 2 AP: une AP qui vas ajouter un element a la fin de la liste Lp et une autre AP pour ajouter un element a la fin de la liste Ln mais on vas ecrire une seule AP qui ajoute un element a la fin d'une liste et on vas lui passer comme parametre Lp ou bien Ln dependamment si la valeur contenue dans l'element est >= 0 ou bien < 0.

Cette AP est une procedure qui ajoute un element d'adresse p a une liste de tete L (sans allouer un nouveau element). Cette procedure ajoutEltFiFo() a 3 parametres: L la tete de la liste ou il faut ajouter, prcd l'adresse du dernier element precedent (s'il y'a un precedent) et l'adresse p de l'element a ajouter.
Si la liste est vide (L = Nil) p est ajoute en tete de liste. Si la liste n'est pas vide (L != Nil) p est ajoute en fin de liste dont l'adresse du dernier est dans prcd.

```
Procedure     ajoutEltFiFo(E/S L: pList, E/S prcd: pList, E/ p: pList)
Debut
   Si (L = Nil) alors
      L <--- p;
   Sinon
      prcd^.svt <--- p;
   Fsi
   prcd <--- p;       /* cet element c'est lui le precedent pour le prochain
*/
   p^.svt <--- Nil;   /* marquer cet element comme etant le dernier */
Fin;
```

Maintenant on va ecrire la procedure qui permet d'eclater la liste L en 2 listes une liste de Nombres positifs et une liste de nompres negatifs.

```
Procedure     CreerListePositifNegatif(E/S L: pList, S/ Lp, Ln: pList)
Var
  p, prcdp, prcdn : pList;
Debut
  Lp <--- Nil;
  Ln <--- Nil;
  Tant Que (L <> Nil)
  Faire
     p <--- L;
     L <--- L^.svt;
     Si (p^.val >= 0)  alors
          /* ajouter a la liste positive */
          ajoutEltFiFo(Lp, prcdp, p);
     Sinon
          /* ajouter a la liste Negative */
          ajoutEltFiFo(Ln, prcdi, p);
     Fsi
  Fait
Fin;
```

EXERCICE 4 :

Soient deux listes L1 et L2 :

1- Ecrire une fonction qui vérifie si L1 et L2 sont identiques (contiennent les mêmes éléments dans le même ordre),

2- Ecrire une fonction qui vérifie si L1 est incluse dans L2 (tous les éléments de L1 se trouvent dans L2, ici l'ordre ne compte pas),

3- Ecrire une fonction qui vérifie si L1 est une sous-liste de L2 (L1 est incluse dans L2 dans le même ordre)

Solution

1- Ecrire une fonction qui vérifie si L1 et L2 sont identiques (contiennent les mêmes éléments dans le même ordre),

```
Fonction    listesIdentiques(L1, L2: pList) : Booleen
Debut
   Tant Que ((L1 <> Nil) et (L2 <> Nil) et (L1.^val = L2^.val))
   Faire
      L1 <--- L1^.svt;
      L2 <--- L2^.svt;
```

```
   Fait

   // les 2 listes L1 et L2 sont identiques si tous les elements de L1
sont egaux a tous les elements
   // de L2 et que L1 et L2 sont arrives a Nil
   Retourner   ((L1 = Nil) et (L2 = Nil));
Fin;
```

2- Ecrire une fonction qui vérifie si L1 est incluse dans L2 (tous les éléments de L1 se trouvent dans L2, ici l'ordre ne compte pas),

```
Fonction   listeL1IncluseL2(L1, L2: pList) : Booleen
Var
   p1, p2: pList;
   incluse: Booleen;
Debut
   /* on suppose au debut que L1 est incluse dans L2 et on vas
parcourir les elements de L1. Des qu'on
        trouve qu'un element de L1 ne se trouve pas dans L2 on
s'arrete.
     */
   incluse <--- Vrai;
   p1 <--- L1;
   Tant Que ((p1 <> Nil) et incluse)
   Faire
      p2 <--- L2;
      // Rechercher l'element courant de L1 (de pointeur p1) a partir du
1er element de L2.
      Tant Que ((p2 <> Nil) et (p2^.val <> p1^.val))
      Faire
          p2 <--- p2^.svt;
      Fait
      Si (p2 = Nil)   alors // p1^.val n'existe pas dans L2
          incluse <--- Faux;
      Sinon
           p1 <--- p1^.svt;   // rechercher le prochain element de L1 dans
L2
      Fsi
   Fait

   Retourner incluse;
Fin;
```

3- Ecrire une fonction qui vérifie si L1 est une sous-liste de L2 (L1 est incluse dans L2 dans le même ordre)

```
Fonction     listeL1SousListeL2(L1, L2: pList) : Booleen
Var
   p1, p2, p2_1: pList;
   sousListe: Booleen;
Debut
   /* on suppose au debut que L1 n'est pas une sous liste de L2 en
mettant sousliste a Faux */
   sousListe ← Faux;
   p2 ← L2;
   Tant Que ((p2 <> Nil) et (sousListe = Faux))
   Faire
      p1 ← L1;
      p2_1 ← p2;
      Tant Que ((p2_1 <> Nil) et (p1 <> Nil) et (p1^.val = p2_1^.val))
      Faire
          p2_1 ← p2_1^.svt;
          p1 ← p1^.svt;
      Fait

      // si p1=Nil cela veut dire qu'on a compare tous les elements de
L1 et qu'ils sont tous egaux aux
      // elements de L2 qui debutent a partir de p2_1
      Si (p1 = Nil) alors
          sousListe ←  Vrai;
      Sinon
          p2 <--- p2^.svt;
      Fsi

   Fait

   Retourner  sousListe;
Fin;
```

EXERCICE 5 :

On veut construire un mot à base des trois lettres A, B et C en respectant les règles suivantes :
Au début (à l'étape 0) le mot est réduit à A. Puis à chaque étape de l'évolution, A est transformé en ABC, B est transformé en C, et C est transformé en A.
Exemple : Etape 0 Etape 1 Etape 2 Etape 3
A → A B C → A B C C A → A B C C A A A B C → …
On veut connaître le mot à l'étape N, N étant donné. En utilisant une liste chaînée, dont chaque cellule contient une lettre (A ou B ou C).

1- Ecrire une procédure permettant d'afficher les éléments d'une liste de caractères.
2- Ecrire un algorithme qui suit l'évolution du mot et affiche le mot construit à chaque étape.

Solution

```
Type
  pListeCar = ^listEltCar;
  listEltCar = Enregistrement
      car : Caractere;
      svt : pListeCar;
  Fin;

Procedure    AfficherListeCar(E/ L:pListeCar)
Debut
  Ecrire("Liste de Caracteres:");
  Tant Que (L <> Nil)  Faire
     Ecrire(L^.car);
     L <-- L^.svt;
  Fait
  Ecrire("Fin de Liste");
Fin;
```

On a besoin des fonctions AjouterCarTete()
et ajouterCarApresPrcd() qui ajoutent un caractere respectivement en tete de liste et apres le precedent.

```
Fonction   ajouterCarTete(L : pListeCar, c: Caractere):pListeCar
Var
   p : pListeCar;
Debut
   Allouer(p)
   p^.car <--- c;
   p^.svt <--- L;

   Retourner p;
Fin;

Fonction    ajouterCarApresPrcd(prcd : pListeCar, c :
Caractere):pListeCar
Var
    p : pListeCar;
Debut
    Allouer(p);
```

```
    p^.car <--- c;
    p^.svt <--- prcd^.svt;
    prcd^.svt <--- p;

    Retourn p;
Fin;

Algorithme  ListeMotABC;
Var
    p, L : pListeCar;
    i, N : Entier;

    Repeter
        Ecrire("Donner N >= 1 ...");
        Lire(N);
    Jusqu'a (N > 0);

    L <--- Nil;
    /* Etape 0: Initialiser liste au caractere 'A' */
    L <-- ajouterCarTete(L, 'A');
    Ecrire("Etape : 0");
    afficherListCar(L);

    Pour i <--- 1 a N
    Faire
        p <--- L;
        Tant Que (p <> Nil)
        Faire
            Cas (p^.car) Vaut
              'A':
                /* 'A' est transforme en 'ABC' en ajoutant 'B' et 'C' a la liste
*/
                p <--- ajouterCarApresPrcd(p, 'B');
                p <--- ajouterCarApresPrcd(p, 'C');

              'B':
                /* 'B' est transformé en 'C' */
                p^.car <--- 'C';

              'C' :
                /* 'C' est transformé en 'A' */
                p^.car <--- 'A';
            FCas
            p <--- p^.svt;
```

```
        Fait
        Ecrire("Etape : ", i);
        afficherListCar(L);
    Fait

Fin.
```

EXERCICE 6 :

Soit FENT un fichier d'entiers.

1- Ecrire une action paramétrée FLISTE permettant de créer une liste chainée L contenant les éléments de FENT dans le même ordre (en FIFO).

2- Soit VAL une valeur entière donnée. Ecrire une action paramétrée permettant de créer une liste POSVAL contenant toutes les positions de la valeur VAL dans la liste L (en LIFO).

3- Soit K un entier positif. Ecrire une action paramétrée DELPOS permettant de supprimer l'élément se trouvant à la Keme position de la liste L

Solution

```
Type
  pList = ^listElt;
  listElt = Enregistrement
      val : Entier;
      svt : pList;
  Fin;
  FichEnt = Fichier de Entier;

Fonction    Fliste(Fent : FichEnt):pList
Var
   L, p, prcd : pList;
   v : Entier;
Debut
   L ← Nil;
   Relire(Fent);
   Si (NoN(FDF(Fent)))  Alors
         Lire(Fent, v);
         Allouer(p);
         L ← p;
         p^.val ←v;
         prcd ← p;
         Tant Que (NoN(FDF(Fent)))  Faire
              Lire(Fent, v);
              Allouer(p);
```

```
            p^.val ← v;
            prcd^.svt ← p;
            prcd ← p;
        Fait
        p^.svt ← Nil;
  Fsi
  Fermer(Fent);
  Retourner L;
Fin;
```

2- Soit VAL une valeur entière donnée. Ecrire une action paramétrée permettant de créer une liste POSVAL contenant toutes les positions de la valeur VAL dans la liste L (en LIFO).

```
Fonction     PosVal(L: pList, Val:Entier):pList
Var
   p, Lval: pList;
   pos : Entier;
Debut
   Lval ← Nil;
   pos ← 0;
   Tant Que (L <> Nil)  Faire
        pos ← pos + 1;
        Si (L^.val = Val)  alors
             Allouer(p);
             p^.val ← pos;
             p^.svt ← Lval;
            Lval ← p;
         Fsi
         L ← L^.svt;
    Fait
    Retourner  Lval;
Fin;
```

3- Soit K un entier positif. Ecrire une action paramétrée DELPOS permettant de supprimer l'élément se trouvant à la Keme position de la liste L

```
Fonction    DelPos(L:pList, K:Entier):pList
Var
  p, prcd: pList;
  i : Entier;
Debut
  p ← L;
```

```
  i ← 1;
  prcd ← Nil;
  Tant Que ((p <> Nil) et (i <> K))
  Faire
      prcd ← p;
      p ← p^.svt;
      i ← i + 1;
  Fait
  Si (p <> Nil) alors
     Si (p = L) // Si (K = 1) ou (prcd = Nil)
         L ← L^.svt;
     Sinon
         prcd^.svt ← p^.svt;
     Fsi
     Liberer(p);
  Fsi
  Retourner L;
Fin;
```

EXERCICE 8:

Soit L une liste d'entiers. Ecrire une fonction qui renvoie l'élément se trouvant au milieu de la liste en utilisant un seul parcours.

```
Type
  pList = ^listElt;
  listElt = Enregistrement
     val : Entier;
     svt : pList;
  Fin;
Fonction    MilieuList(L: pList):Entier
Var
  p1, p2: pList;

  p1 <--- L;
  p2 <--- L;

  /* Utiliser 2 pointeurs p1 et p2: Faire avancer p1 d'une position et
p2 de 2 positions
     Lorsque p2 arrive en fin de liste, p1 sera au milieu.
   */
  Tant Que ((p2 <> Nil) et (p2->svt <> Nil))
  Faire
     p1 <--- p1^.svt;
     p2 <--- p2^.svt;
```

```
      p2 <--- p2^.svt;
   Fait

   Retourner p1^.val;
Fin;
```

EXERCICE 9 :

Soit T un tableau de 26 listes de chaînes de caractères. La liste 1 contient des mots commençant par la lettre ‘A’, la liste 2 contient des mots commençant par la lettre ‘B’…etc.
Déclarer T et écrire une fonction qui vérifie l’existence d’un mot M dans la structure.

```
Type
  pList = ^listElt;
  listElt = Enregistrement
      val : Chaine;
      svt : pList;
  Fin;

Fonction   RechercherMot(T:Tableau[1..26] de pList, M:Chaine):
Booleen
Var
   p: pList;
   existe: Booleen;
Debut
   existe <-- Faux;
   // Si le 1er caractere du mot M n'est pas compris entre 'A' et 'Z'
alors le mot ne peut pas exister
   // dans les liste de T
   Si ((M[1] >= 'A') et (M[1] <= 'Z'))   alors // dans ce cas rechercher
mot M dans T
        p <-- T[(M[1] - 'A') + 1];  // p contient la tete de liste ou mot M
peut se trouver
        Tant Que ((p <> Nil) et (existe = Faux))  Faire
            Si (p^.val = M) alors
                existe <-- Vrai;
            Sinon
                p <-- p^.svt;
            Fsi
        Fait
   Fsi
   Retourner existe;
Fin;
```

EXERCICE 10 :

Soient deux listes (L1 et L2) de valeurs entières positives :

1) Donner les déclarations des listes ;

2) Ecrire une action paramétrée permettant de déplacer (sans allocation ni libération) les valeurs paires de L1 vers L2, et, de déplacer les valeurs impaires de L2 vers L1 ;

```
Procedure DeplacerPairImpair(E/S L1, L2: pList)
Var
   p1, prec1, p1Svt: pList;
   p2, prec2, p2Svt: pList;
Debut

   p1 <--- L1;
   p2 <--- L2;
   prec2 <--- Nil;

   /* Deplacer les valeurs paires de L1 vers L2 */
   Tant Que (p1 <> Nil)
   Faire
      Si ((p1^.val mod 2) <> 0)  alors
        /* Si valeur n'est pas paire mettre p1 dans prec1 et avancer p1
vers suivant */
         prec1 <--- p1;
         p1 <--- p1^.svt;
      Sinon
        /* Valeur paire enlever p1 de liste L1 */
        Si (p1 = L1) alors
           /* p1 est le 1er element de L1 alors avancer L1 */
           L1 <--- L1^.svt;
           p1Svt <--- L1;  /* sauvegarder dans p1Svt prochain elt de L1
a traiter */
        Sinon
           /* Mettre a jour le suivant de prec1 */
           prec1^.svt <--- p1^.svt;
           p1Svt <--- p1^.svt;
        Fsi

        /* p1 a ete enleve de L1 maintenant l'ajouter a la liste L2 */
        /* Si prec2 est Nil ajouter p1 en tete de liste de L2 sinon
          l'ajouter apres prec2 pour garder le meme ordre dans L2 */
         Si (prec2 = Nil) alors
           p1^.svt <--- L2;
```

```
            L2 <--- p1;
            prec2 <--- p1;
          Sinon
            p1^.svt <--- prec2^.svt;
            prec2^.svt <--- p1;
            prec2 <--- p1;
          Fsi
          p1 <--- p1Svt;  /* Traiter le prochain elt de L1 */
        Fait

        /* Deplacer les nombres impairs de L2 vers L1 */
        prec1 <--- Nil;
        Tant Que (p2 <> Nil)
        Faire
            Si ((p2^.val mod 2) = 0)  alors
              /* Si valeur n'est pas impaire mettre p2 dans prec2 et
avancer p2 vers suivant */
              prec2 <--- p2;
              p2 <--- p2^.svt;
            Sinon
              /* Valeur est impaire enlever p2 de L2 */
              Si (p2 = L2) alors
                /* p2 est le 1er element de L2 alors avancer L2 */
                L2 <--- L2^.svt;
                p2Svt <--- L2;  /* sauvegarder dans p2Svt prochain elt de
L2 a traiter */
              Sinon
                /* Mettre a jour le suivant de prec2 */
                prec2^.svt <--- p2^.svt;
                p2Svt <--- p2^.svt;
              Fsi

              /* p2 a ete enleve de L2 maintenant l'ajouter a la liste L1 */
              /* Si prec1 est Nil ajouter p2 en tete de liste de L1 sinon
                 l'ajouter apres prec1 pour garder le meme ordre dans L1
*/
              Si (prec1 = Nil) alors
                p2^.svt <--- L1;
                L1 <--- p2;
                prec1 <--- p2;
              Sinon
                p2^.svt <--- prec1^.svt;
                prec1^.svt <--- p2;
                prec1 <--- p2;
```

```
            Fsi
            p2 <--- p2Svt;  /* Traiter le prochain elt de L2 */
        Fait
Fin;
```

EXERCICE 11 :
Un pharmacien souhaite traiter les informations concernant son stock de médicaments par ordinateur. On vous propose de représenter ces informations sous forme de liste linéaire chainée où chaque élément contient le libellé d'un médicament, la quantité disponible (nombre de boites) et le prix unitaire.
1. Donner les structures de données nécessaires a la représentation de ce stock (voir schéma).
2. Ecrire la procédure Vendre (Med, NbBoites) permettant de retirer, si possible, 'NbBoites' du médicament 'Med' du stock. (Il faut supprimer du stock le médicament dont la quantité atteint 0).
3. Ecrire la procédure Acheter (Med, NbBoites, Prix) permettant au pharmacien d'alimenter son stock par'NbBoites' du médicament 'Med' ayant le prix unitaire 'Prix' DA. On considère qu'un médicament prenne toujours le nouveau prix. Si le médicament n’existe pas, il faut l’insérer.
4. Ecrire la fonction ValStock permettant de calculer la valeur des médicaments dans le stock.

1-

```
Type
  Pmedicament = ^Medicament ;
  Medicament = Enregistrement
     libele :chaine[50] ;
     qt :entier ;   // quantite
     prix :reel ;   // prix
     svt : Pmedicament ;
  Fin ;
```

2-

```
Procedure Vendre(E/S L :Pmedicament ;E/ Med :chaine[50] ; E/ Nb
:entier ; S/ venteFaite :booleen) ;
Var
  p, prcd :Pmedicament ;
Debut
  venteFaite <-- Faux ;
  Si (L <> Nil) alors
     p <-- L;
```

```
    // rechercher  Med
    Tantque ( (p <> Nil) et (p^.libele <> Med)) Faire
       prcd <-- p ;
       p <-- p^.svt ;
    Fait ;
    Si (p <> Nil) alors
       // on a trouvé Med, tester si Nb nombre de boites disponibles
       Si (p^.qt >= Nb) alors
          venteFaite <-- Vrai ;
          p^.qt <-- p^.qt - Nb ;
         // si quantite devient nulle, alors supprimer p
         Si (p^.qt = 0) alors
              Si (p = L) alors // cas supprimer tete de liste
                    L <-- L^.svt ;
              Sinon
                    prcd^.svt <-- p^.svt;  // cas supprimer apres precedent
              Fsi
              Libérer(p) ;
         Fsi
       Fsi
    Fsi
  Fsi
Fin ;

3-
Fonction Acheter(L:Pmedicament, Med :chaine[50], Nb :entier, prix
:reel) : PMedicament
Var
  p, p1, prcd :Pmedicament ;
Debut
  Si (L = Nil)  alors // liste de medicaments est vide
     Allouer(p1) ;
     p1^.libele <-- Med ;
     p1^.qt <-- Nb ;
     p1^.prix <-- prix ;
     p1^.svt <-- Nil;
     L <-- p1;
  Sinon // Rechercher Med  dans la liste de medicaments L
     p <-- L ;
     Tantque ( (p <> Nil) et (p^.libele <> Med)) Faire
        prcd <-- p ;
        p <-- p^.svt ;
     Fait
     Si (p <> Nil)  alors
```

```
        // on a trouvé Med, on met à jour les données
        p^.qt <-- p^.qt + Nb ;
        p^.prix <-- prix;
    Sinon // Med n'existe pas, on le crée à la fin, après le dernier
(prcd)
        Allouer(p1) ;
        p1^.libele <-- Med ;
        p1^.qt <-- Nb ;
        p1^.prix <-- prix ;
        p1^.svt <-- Nil;
        prcd^.svt <-- p1;
    Fsi
  Fsi

  Retourner L;
Fin ;
```

4-

```
Fonction ValStock(L :Pmedicament) :reel ;
Var
  val : Reel;
Debut
  val <-- 0;
  Tantque (L <> Nil)  Faire
     val <-- val + (L^.qt * L^.prix) ;
     L <-- L^.svt ;
  Fait

  Retournet val;
Fin ;
```

EXERCICE 12 :

Soit L une liste de caractères constituant des mots (un mot est une suite de caractères ne contenant pas de blanc) séparés par un seul caractère blanc (espace).
Ecrire une procédure qui inverse les mots de la liste L sans création d'une nouvelle liste.

```
Type
 carList = ^Element;
 Element = Enregistrement
    car : Caractere;
    svt : carList;
```

```
 Fin;;
```

Si on a la liste suivante:
Liste:
'a' ->'r' ->'b' ->'r' ->'e' ->' ' ->'c' ->'h' ->'a' ->'i' ->'s' ->'e' ->' ' ->'t' ->'a' ->'b' ->'l' ->'e' ->' ' ->'m' ->'i' ->'r' ->'o' ->'i' ->'r' ->Nil
on devrait obtenir la Liste comme:
'e' ->'r' ->'b' ->'r' ->'a' ->' ' ->'e' ->'s' ->'i' ->'a' ->'h' ->'c' ->' ' ->'e' ->'l' ->'b' ->'a' ->'t' ->' ' ->'r' ->'i' ->'o' ->'r' ->'i' ->'m' ->Nil

```
Fonction    InverserMotsListe(L: carList): carList
Var
   LN, p, prcd, psvt, prem, ppBlanc: carList;
Debut
    p <--- L;
    prcd <-- Nil;
    prem <-- L;
    // Traiter le 1er mot a part
    Tant Que ((p <> Nil) et (p^.car <> ' '))  // inverser les caracteres du 1er mot
         psvt <--- p^.svt;
         p^.svt <--- prcd;
         prcd <--- p;
         p <--- psvt;
    Fait
    // la nouvelle tete de liste est l'ancien dernier caractere du 1er mot qui
   // est devenu le 1er caractere du 1er mot apres inversion
    LN <-- prcd;
    Si (p <> Nil) alors // s'il y'a d'autres mots alors p contient l'@ d'un blanc
       // l'ancien 1er caractere du 1er mot son suivant est un blanc car il est devenu le dernier
       // caractere du 1er mot
       prem^.svt <- p;
       prcd <-- p;  // prcd contient l'@ d'un blanc

       Tant Que (p <> Nil)
       Faire
          p <-- p^.svt; // avancer vers le 1er caractere du prochain mot
          pBlanc <-- prcd;  // sauvegarder l'@ du blanc
          prem <-- p;  // sauvegarder @ de l'ancien 1er caractere du mot

          /* Inverser les caracteres du mot courant */
```

```
            Tant Que ((p <> Nil) et (p^.car <> ' '))
            Faire
                psvt <--- p->svt;
                p->svt <--- prcd;
                prcd <--- p;
                p <--- psvt;
            Fait

            // p contient l'@ d'un blanc ou bien Nil (dans le cas du dernier
mot)
            // l'ancien 1er caractere du mot son suivant est un blanc car il
est devenu le dernier
           // caractere du mot ou bien son suivant est Nil dans le cas du
dernier mot
            prem^.svt <-- p;

            // le blanc doit avoir so suivant l'ancien dernier caractere qui
est devenu  le 1er caractere du mot
            pBlanc^.svt <-- prcd;
            prcd <-- p;  // mettre dans prcd l'@ du blanc (si ce n'est pas le
dernier mot)
        Fait
      Fsi

      Retourner LN;
Fin;
```

EXERCICE 13 :

Dans cet exercice, on se propose de développer un module permettant de manipuler des polynômes creux. Un polynôme creux est un polynôme contenant très peu de monômes non nuls.
Exemple : P(x) = 5.6 x1280 + 0.8 x – 9 contient 1281 termes dont 3 seulement sont non nuls. Chaque monôme est décrit par un enregistrement de type Tmonome comportant les 3 champs suivants :

¬ Deg : entier représentant le degré du monôme ;
¬ Coef : réel représentant le coefficient du monôme ;
¬ Suiv : pointeur sur le monôme suivant.

La liste représentant le polynôme sera triée par ordre de degré décroissant. Une liste vide (Nil) correspond au polynôme zéro ;
1- Ecrire une procédure Ajouter qui ajoute à un polynôme (Pol) la valeur d'un monôme défini par son degré (Deg) et son coefficient (Coef).

2- Ecrire les procédures Somme et Produit qui réalisent respectivement la somme et le produit de deux polynômes Pol1 et Pol2.
3- Ecrire une fonction Valeur qui calcule la valeur du polynôme pour une valeur val de la variable x.
4- Ecrire une procédure Derive qui détermine la dérivée DPol d'un polynôme Pol.
5- Ecrire une procédure Primitive qui détermine la primitive PPol d'un polynôme Pol, sachant que PPol(0)=1.

```
Type
   Tmonome=^Monome ;
   Monome=Enregistrement
      coef :reel ;
      deg :entier ;
      svt :Tmonome ;
   Fin ;
1-
Fonction Ajouter(pol :Tmonome ; M :Tmonome) : Tmonome
Var
  p, prcd :Tmonome ;
Debut

 Si (M^.coef <> 0) alors // si coefficient de M = 0 alors rien a faire
   // si liste Pol vide ou Monome degree > degree du 1er monome
alors ajouter en tete du polynome
   Si ((pol = Nil) OU (M.deg > pol.deg)) alors
     pol <-- M ;
     M^.svt <-- pol;
   Sinon /*recherche de position ou ajouter le monome
     p <-- pol^.svt;
     prcd <-- pol;
     Tantque ((p <> Nil) et (p^.deg > M^.deg))  Faire
        prcd <-- p;
        p <-- p^.svt;
     Fait
     Si ((p <> Nil) et (p^.deg = M^.deg))  alors
        // si monome de meme degree alors ajouter les coefficients
        p^.coef <-- p^.coef + M^.coef;
        // liberer M car on n'a plus besoin
        Liberer(M);
        // si coefficient de p est = 0 alors supprimer p
        Si (p^.coef = 0) alors
           prcd^.svt <-- p^.svt;
```

```
          Liberer(p);
        Fsi
      Sinon
        // ajouter Monome M apres prcd
        prcd^.svt <-- M;
        M^.svt <-- p;
      Fsi
    Fsi
  Fsi

  Retourner pol;
Fin;
```

2-

```
Fonction Somme(pol1,pol2 :Tmonome) : TMonome
Var
  pol3, M :Tmonome ;
Debut
  /*Ajouter tous les monômes de Pol1 à Pol3== Duplication de Pol1
Dans Pol3
  pol3 <-- Nil ;
  Tantque (pol1 <> Nil)  Faire
     Allouer(M) ;
     M^.deg <-- pol1^.deg ;
     M^.coef <-- pol1^.coef ;
     Ajouter(pol3,M) ;
     pol1 <-- pol1^.svt ;
  Fait
  /*Ajouter tous les monômes de Pol2à Pol3
  Tantque (pol2 <> Nil)  Faire
     Allouer(M) ;
     M^.deg <-- pol2^.deg ;
     M^.coef <-- pol2^.coef ;
     Ajouter(pol3,M) ;
     pol2 <-- pol2^.svt ;
   Fait
Fin;

Fonction Produit(pol1,pol2 :Tmonome) : Tmonome
Var
  pol3, M,p :Tmonome ;
Debut
  pol3 <-- Nil ;
  Si ((pol1 <> Nil) et (pol2 <> Nil))  alors
```

```
    // Multiplier chaque monome de pol2 par pol1 et ajouter a pol3
    Tantque (pol2 <> Nil)  Faire
        p <-- pol1 ;
        Tantque (p <> Nil)  Faire
            Allouer(M) ;
            M^.deg <-- pol2^.deg + p^.deg ;
            M^.coef <-- pol2^.coef * p^.coef ;
            Ajouter(pol3,M) ;
            p←p^.svt ;
        Fait
        pol2 <-- pol2^.svt ;
    Fait
  Fsi

  Retourner pol3;
Fin ;

3-
Fonction Valeur(pol :Tmonome ,X :reel ):reel ;
Var
  i : entier ;
  Px, val : reel ;
Debut
  val <-- 0;
  Tantque (pol <> Nil)  Faire
      Px <-- 1 ;
      // calcul de puissance de X
      Pour i <-- 1 a pol^.deg Faire
          Px <-- Px * X ;
      Fait
      val <-- val + pol^.coef  *  Px ;
      pol <-- pol^.svt ;
  Fait

  Retourner val;
Fin

4-
Fonction Derive(pol :Tmonome) : Tmonome
Var
  dpol, M,p :Tmonome ;
Debut
  dpol <-- Nil ;
  Si (pol <> Nil) alors
```

```
    // créer la le premier monôme de DPol
    Si (pol^.deg <> 0) alors
       Allouer(dpol) ;
       dpol^.coef <-- pol^.coef * pol^.deg ;
       dpol^.deg <-- pol^.deg - 1 ;
       p <-- dpol
    Fsi
    pol <-- pol^.svt ;
    // créer les autres monômes de dpol
    Tantque (pol <> Nil)  Faire
        Si (pol^.deg <> 0) alors
           Allouer(M)) ;
           M^.coef <-- pol^.coef * pol^.deg ;
           M^.deg <-- pol^.deg - 1 ;
           p^.svt <-- M ;
           p <-- M ;
        Fsi ;
        pol <-- pol^.svt ;
     Fait ;
     Si (dpol <> Nil) alors
         p^.svt <-- Nil
     Fsi
  Fsi

  Retourner dpol;
Fin ;

5-
Fonction Primitive(pol :Tmonome) : Tmonome
Var
  ppol, M,p :Tmonome ;
Debut
  ppol <-- Nil ;
  Si (pol <> Nil) alors
    // créer la le premier monôme de PPol
    Allouer(ppol) ;
    ppol^.coef <-- pol^.coef/(pol^.deg + 1) ;
    ppol^.deg <-- pol^.deg + 1 ;
    p <-- ppol ;
    pol <-- pol^.svt ;
    // créer les autres monômes de ppol
    Tantque (pol <> Nil)  Faire
       Allouer(M)) ;
       M^.coef <-- pol^.coef/(pol^.deg + 1) ;
```

```
        M^.deg <-- pol^.deg + 1 ;
        p^.svt <-- M ;
        p <-- M ;
        pol <-- pol^.svt ;
     Fait
   Fsi
   // créer la constante
   Allouer(M) ;
   M^.deg <-- 0 ;
   M^.coef <-- 1 ;
   M^.svt <-- Nil ; //PPol(0)=1
   Si (ppol <> Nil) Alors
      p^.svt <-- M;
   Sinon
      ppol <-- M;
   Fsi

   Retourner ppol;
Fin ;
```

EXERCICE 14 :

Soit L une liste d'entiers.

1- Ecrire une procédure DETACHE qui renvoie l'adresse du minimum de la liste L et le détache de la liste sans le supprimer.

2- En utilisant la procédure DETACHE, écrire une procédure TRIER qui trie la liste L dans un ordre décroissant (sans création de nouvelle liste).

```
Procedure    Detacher(E/S   L, pMin: pList)
Var
   p, prec, prcdMin: pList;
Debut
   Si (L = Nil) alors    /* Si liste est vide alors pMin est Nil */
          pMin <--- Nil;
   Sinon
          /* Supposer que le 1er element de la liste est le minimum
              et commencer a comparer a partir du 2eme element de la
liste.
            */
          pMin <--- L;
          p <--- L^.svt;
          prec <--- L;
          Tant Que (p <> Nil)
          Faire
```

```
                Si (p^.val < pMin^.val) alors
                    pMin <--- p;
                    prcdMin <-- prec;
                Fsi
                prec <--- p;
                p <--- p^.svt
            Fait

            /* Detacher pMin de L */
            Si (pMin = L) alors
                L <--- L^.svt;
            Sinon
                prcdMin^.svt <--- pMin^.svt;
            Fsi
            pMin^.svt <--- Nil;
    Fsi
Fin;

Procedure    Trier(E/S L, LT: pList)
Var
  pMin: pList;
Debut
  LT <--- Nil;
  Tant Que (L <> Nil)
  Faire
        Detacher(L, pMin);            /* appeler Detacher() pour avoir le
min */
        pMin^.svt <-- LT;             /* ajouter ce pMin en tete de liste:
son suivant est l'ancien premier*/
        LT <--- pMin;                 /* ce pMin est celui qui devient le 1er
(liste decroissante)*/
  Fait
Fin;
```

EXERCICE 15 :

Soient deux listes L1 et L2 :

1- Ecrire une fonction qui vérifie si L1 et L2 sont disjointe (L1 ∩ L2 = Ø).

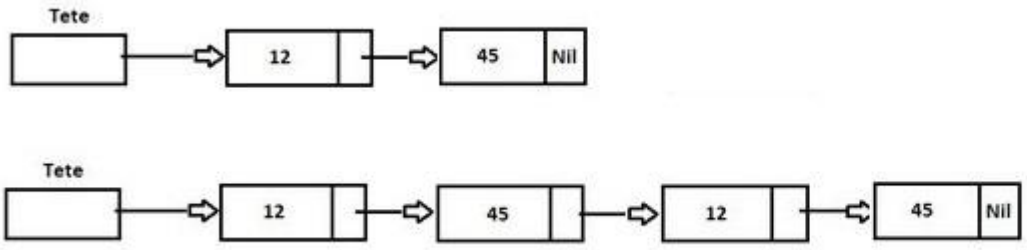
2- Ecrire une fonction qui vérifie si L1 est préfixe de L2 (L2 commence par L1).

```
Fonction  listeL1L2Disjointes(L1, L2: pList) : Booleen
Var
   p1, p2: pList;
```

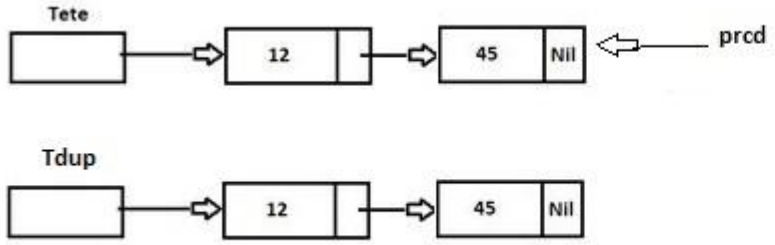
```
   disjoint: Booleen;
Debut
   disjoint <--- Vrai;
   p1 <--- L1;
   Tant Que ((p1 <> Nil) et disjoint)
   Faire
      p2 <--- L2;
      Tant Que ((p2 <> Nil) et (p2^.val <> p1^.val))
      Faire
          p2 <--- p2^.suivant;
      Fait
      Si   (p2 <> Nil)  alors
           disjointes <--- Faux;
      Sinon
          p1 <--- p1^.suivant;
      Fsi
   Fait

   Retourner disjoint;
Fin;

Fonction  listeL1PrefixeL2(L1, L2: pList) : Booleen
Debut
   Tant Que ((L1 <> Nil) et (L2 <> Nil) et (L1^.val = L2^.val))
   Faire
      L1 <-- L1^.svt;
      L2 <-- L2^.svt;
   Fait
   // L2 commence par L1 cela veut dire qu'on a parcourut tous les
elements de L1 dans le Tant Que
   // et tous les elements de L1 sont egaux aux elements de L2 du
meme rang et qu'on est arrive a L1=Nil
   Si (L1 = Nil) alors
       Retourner Vrai;
   Sinon
       Retourner Faux;
   Fsi


Fin;
```

Exercice

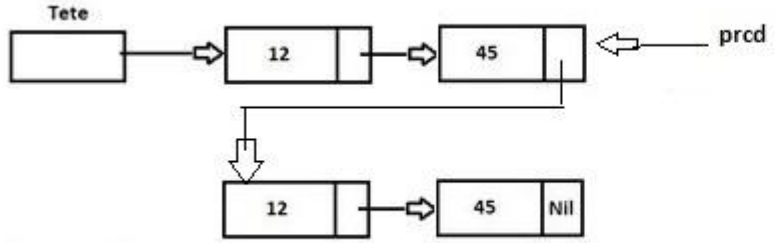
Ecrire une AP qui duplique le contenu d'une liste entiere.
Exemple:



On vas d'abord parcourir la liste Tete et creer la liste Tdup qui est la duplique de la liste Tete (liste FIFO car on doit garder le meme ordre des elements). Lorsqu'on parcourt la liste tete on doit garder l'adresse du dernier element dans prcd.



Ensuite on attache la fin de la liste tete avec la liste Tdup et la liste tete devient:



Notre AP est une procedure car la tete de Liste Tete ne sera pas modifie.

```
Procedure DupliqueListe(E/ Tete: pList)
Var
   p, pDup, prcdDup, prcd, Tdup: pList;
Debut
   // Creer la liste Tdup a partir de liste Tete (Liste FiFo)
   Si (Tete <> Nil)  alors
      // dupliquer le 1er element de liste Tete
      Allouer(pDup);
      pDup^.val ← Tete^.val;
      Tdup ← pDup;
      prcdDup ← Tdup;
      p ← Tete^.svt;
      prcd ← Tete;
```

```
      // dupliquer les autres elements de Tete
       Tant Que (p <> Nil)  Faire
            Allouer(pDup);
           pDup^.val ← p^.val;
           prcdDup^.svt ← pDup;
           prcdDup ← pDup;

           // faire avancer p en gardant le precedent
           prcd ← p;
           p ← p^.svt;
       Fait
       pDup^.svt ← Nil;

       // attacher la fin de la liste Tete avec la liste Tdup
       prcd^.svt ← Tdup;
  Fsi
Fin;
```