

## Exercices des chapitres 9, 10 et 11

### Sommaire

#### Exercices

01-**- Fonction de comptage dans une liste chaînée .....	2
02-**- Fonction de comptage d'occurrences dans une liste chaînée .....	2
03-**-Fonction de vérification d'une liste chaînée triée .....	2
04-**-Procédure d'insertion en tête de liste chaînée .....	2
05-**-Procédure d'insertion en queue de liste chaînée .....	2
06-**- Procédure d'insertion à une position donnée .....	2
07-**- Procédure de suppression d'un élément d'une liste chaînée à une position donnée .....	2
08-**- Fonction de calcul de moyenne des étudiants .....	3
09-**- Procédure de parcours d'une liste circulaire ou anneau .....	5
10-**- Procédure d'insertion d'un élément dans une liste doublement chaînée.....	5
11-***- Procédure de suppression d'un élément dans une liste doublement chaînée .....	5
12-***- Procédure de suppression d'un étudiant dans une liste doublement chaînée .....	5
13-***- Procédure d'insertion d'un étudiant dans une liste doublement chaînée triée .....	5

#### Corrigés

01-**- Fonction de comptage dans une liste chaînée .....	7
02-**- Fonction de comptage d'occurrences dans une liste chaînée .....	7
03-**-Fonction de vérification d'une liste chaînée triée .....	8
04-**-Procédure d'insertion en tête de liste chaînée .....	8
05-**-Procédure d'insertion en queue de liste chaînée .....	9
06-**- Procédure d'insertion à une position donnée .....	10
07-**- Procédure de suppression d'un élément d'une liste chaînée à une position donnée .....	11
08-**- Procédure de calcul de moyenne des étudiants .....	12
09-**- Procédure de parcours d'une liste circulaire ou anneau .....	12
10-**- Procédure d'insertion d'un élément dans une liste doublement chaînée.....	13
11-***- Procédure de suppression d'un élément dans une liste doublement chaînée .....	14
12-***- Procédure de suppression d'un étudiant dans une liste doublement chaînée .....	16
13-***- Procédure d'insertion d'un étudiant dans une liste doublement chaînée triée .....	17

On considérera dans les exercices, sauf cas contraire une liste chaînée de ce type :

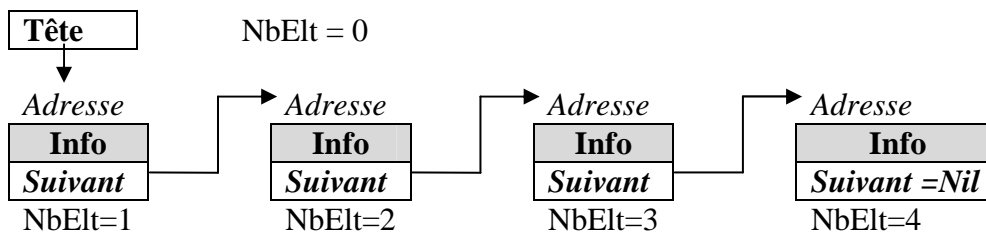
Type Liste =  $\wedge$ Cellule

Type Cellule = Structure

Info : chaîne de caractères

Suivant : Liste

Fin structure



### 01-\*\*- Fonction de comptage dans une liste chaînée

Ecrire une fonction qui renvoie le nombre d'éléments d'une liste chaînée.

### 02-\*\*- Fonction de comptage d'occurrences dans une liste chaînée

Ecrire une fonction qui renvoie le nombre d'éléments d'une liste chaînée ayant une valeur donnée (champ Info).

### 03-\*\*-Fonction de vérification d'une liste chaînée triée

Ecrire une fonction qui vérifie si une liste chaînée est triée par valeurs croissantes du champ Info.

### 04-\*\*-Procédure d'insertion en tête de liste chaînée

Ecrire une procédure qui insère un nouvel élément en tête d'une liste chaînée.

### 05-\*\*-Procédure d'insertion en queue de liste chaînée

Ecrire une procédure qui insère un nouvel élément en queue d'une liste chaînée.

### 06-\*\*- Procédure d'insertion à une position donnée

Ecrire une procédure qui insère un nouvel élément de sorte qu'il se trouve à une position donnée dans la liste. La position est un entier et correspond au numéro du futur élément dans la liste. Le premier élément porte le numéro 1.

### 07-\*\*- Procédure de suppression d'un élément d'une liste chaînée à une position donnée

Ecrire une procédure qui supprime un élément d'une liste chaînée à une position donnée.

## 08-\*\*- Fonction de calcul de moyenne des étudiants

Le département dans lequel vous êtes inscrit souhaite gérer les notes de ses étudiants. Les étudiants ont pour identifiant leur numéro d'étudiant. Ils ont un nom et un prénom. Ces informations sont stockées dans une liste chaînée dont chaque élément comporte aussi un champ *moy* pour la moyenne de l'étudiant et un champ *eval* qui est un pointeur sur sa liste de notes. La liste de notes de chaque étudiant est aussi une liste chaînée dont la tête est le champ *eval* de la cellule de l'étudiant.

On dispose des déclarations suivantes :

Types :

```
Ch10      = Chaîne de 10 caractères
Ch30      = Chaîne de 30 caractères
Ent       = entier
Nb        = réel
Pe        = ^Etudiant
Pn        = ^Note
```

```
Etudiant  = Structure
  no      : Ch10
  nom     : Ch30
  prenom  : Ch30
  moy     : Nb
  eval    : Pn
  suivant : Pe
          fin Structure
```

```
Note      = Structure
  note    : Nb
  coeff   : Ent
  suivant : Pn
          fin Structure
```

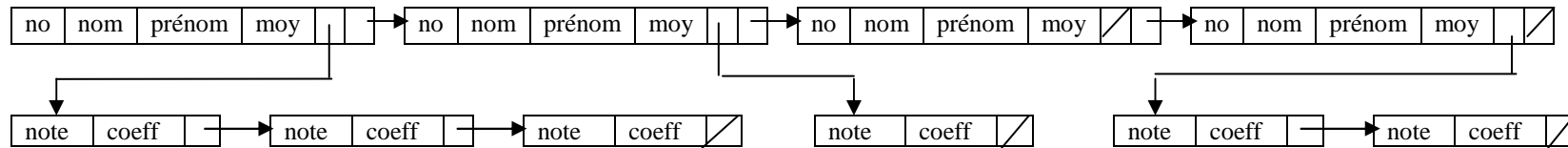
Faire un schéma de cette structure et vérifier à la page suivante où elle est représentée.

On suppose que tous les champs de la liste des étudiants sont remplis sauf le champ *moy*. On suppose que toutes les notes des étudiants et tous les coefficients sont remplis

Écrire une procédure *moyennesEtudiants* qui parcourt la liste des étudiants, et qui calcule et met à jour le champ *moy* de chaque étudiant à l'aide de la liste des notes sur laquelle pointe le champ *eval*. La procédure *moyennesEtudiants* prend en paramètre la tête de la liste des étudiants.

.

On peut représenter cette structure par la figure ci-dessous :



*Remarques :*

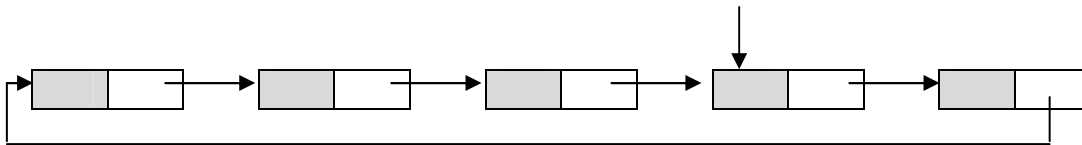


Cette notation équivaut à Nil

Il se peut qu'un étudiant n'ait pas encore de note. C'est le cas du 3<sup>ème</sup> étudiant de la liste de l'exemple ci-dessus. Le pointeur *eval* est égal à Nil.

## 09-\*\*- Procédure de parcours d'une liste circulaire ou anneau

Les listes circulaires ou anneaux sont des listes linéaires dans lesquelles le dernier élément pointe sur le premier. Il n'y a donc ni premier, ni dernier. L'anneau connaît l'adresse d'un élément pour parcourir tous les éléments de la liste.



Ecrire une procédure *traite\_liste* qui « traite » chaque élément de la liste en appelant une procédure *traiter* qui aura comme paramètre un pointeur sur l'élément courant à traiter. La procédure *traite\_liste* prend en paramètre un pointeur sur un élément quelconque de la liste. On considère que la liste contient au moins un élément (**liste non vide**).

## 10-\*\*- Procédure d'insertion d'un élément dans une liste doublement chaînée

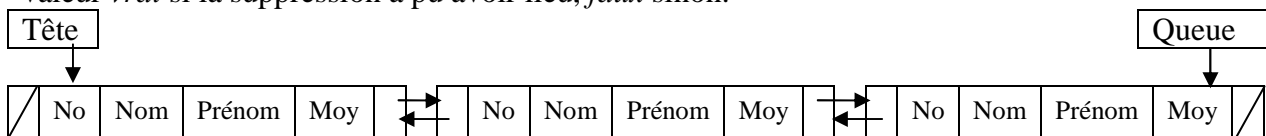
Ecrire une procédure insérant un nouvel élément dans une liste doublement chaînée, avant l'élément de la liste ayant une valeur donnée (dans sa zone info). On dispose d'une fonction PtV(Tête, Val) qui renvoie l'adresse du premier élément de la liste qui porte la valeur "val", ou Nil si cette valeur n'existe pas.

## 11-\*\*\*. Fonction de suppression d'un élément dans une liste doublement chaînée

Ecrire une procédure supprimant, dans une liste doublement chaînée, un élément ayant une valeur donnée (dans sa zone info). Dans les paramètres de la procédure, il doit y avoir un paramètre booléen qui aura comme valeur *vrai* si la suppression a pu avoir lieu, *faux* sinon.

## 12-\*\*\*. Fonction de suppression d'un étudiant dans une liste doublement chaînée

Ecrire une procédure supprimant, dans une liste doublement chaînée, un étudiant ayant un numéro donné. Dans les paramètres de la procédure, il doit y avoir un paramètre booléen qui aura comme valeur *vrai* si la suppression a pu avoir lieu, *faux* sinon.



## 13-\*\*\*. Procédure d'insertion d'un étudiant dans une liste circulaire triée

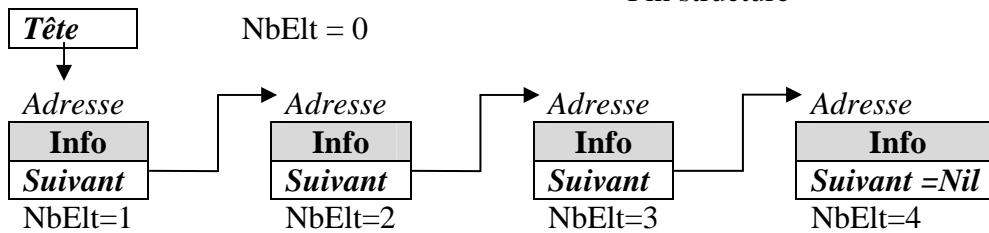
Ecrire une procédure insérant un étudiant dans une liste doublement chaînée, qui doit restée triée par ordre croissant du *No*. Il faudra donc insérer le nouvel étudiant juste avant le premier numéro d'étudiant supérieur à celui que l'on souhaite ajouter dans la liste. On ne demande pas de gérer les éventuels doublons de numéros.

La liste a le même schéma que dans l'exercice 12.

# CORRIGES

On considérera dans les exercices, sauf cas contraire une liste chaînée de ce type :

Type Liste =  $\wedge$ Cellule  
 Type Cellule = Structure  
     Info : chaîne de caractères  
     Suivant : Liste  
 Fin structure



## 01-\*\*- Fonction de comptage dans une liste chaînée

```

fonction cpterEltListeChaine(Tête : Liste) : entier
/*
* Comptage des éléments d'une liste chaînée passée en paramètre
*/
Variables :
    nbElt : entier /* nombre d'éléments de la liste */
    pliste : Liste /* pointeur de parcours de la liste */
Début
    /* Initialisation des variables */
    nbElt ← 0
    pliste ← Tête
    /* Boucle de parcours des éléments de la liste passée en paramètre */
    tantque pliste ≠ Nil
        nbElt ← nbElt + 1 /* on incrémente nbElt */
        pliste ← pliste^.suivant /* on passe à l'élément suivant */
    fintantque
    retourner(nbElt)
Fin
  
```

## 02-\*\*- Fonction de comptage d'occurrences dans une liste chaînée

```

fonction cpterOccEltListeChaine(Tête : Liste, val : chaîne) : entier
/*
* Comptage des occurrences d'une valeur d'une liste chaînée passée en paramètre
*/
Variables :
    nbOcc : entier /* nombre d'occurrences trouvées dans la liste */
    pliste : Liste /* pointeur de parcours de la liste */
Début
    nbOcc ← 0 /* Initialisation des variables */
    pliste ← Tête
    /* Boucle de parcours des éléments de la liste passée en paramètre */
    tantque pliste ≠ Nil
        si pliste^.info = val alors
            nbOcc ← nbOcc + 1 /* on incrémente nbOcc */
        fin si
        pliste ← pliste^.suivant /* on passe à l'élément suivant */
    fintantque
    retourner(nbOcc)
fin
  
```

### 03-\*\*-Fonction de vérification d'une liste chaînée triée

Ecrire une fonction qui vérifie si une liste chaînée est triée en ordre croissant du champ *Info*.

```

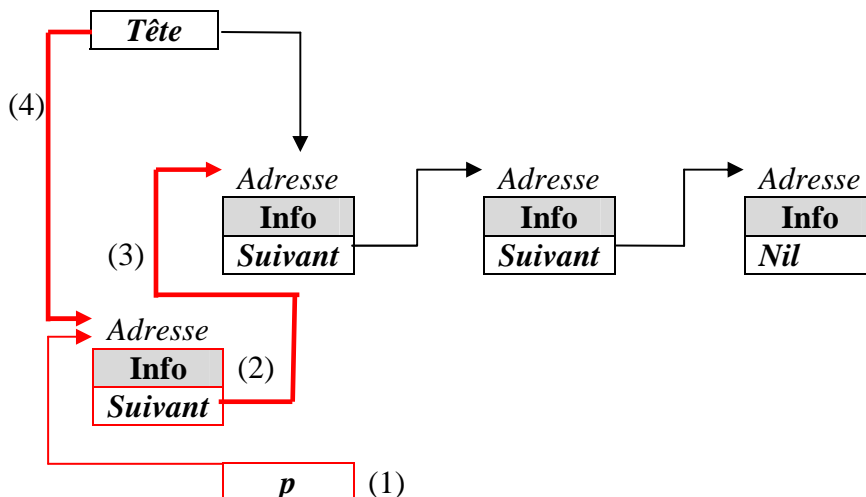
fonction estTrie(Tête : Liste) : booléen
/*
* renvoie vrai si la liste est triée, faux sinon
*/
Variables :
    trie : booléen           /* indicateur pour l'ordre */
    pliste : Liste           /* pointeur de parcours */
Début
    /* Initialisation des variables */
    trie ← vrai              /* la liste est considérée triée au départ */
    pliste ← Tête
    /* si la liste n'est pas vide */
    si pliste ≠ Nil alors
        /* Boucle de parcours des éléments */
        tantque pliste^.suivant ≠ Nil et trie faire
            si pliste^.info ≤ pliste^.suivant^.info alors
                pliste ← pliste^.suivant /* on passe au suivant */
            sinon
                trie ← faux              /* la liste n'est pas triée */
            finsi
        fintantque
    finsi
    retourner(trie)
Fin

```

### 04-\*\*-Procédure d'insertion en tête de liste chaînée

**Actions à mener :**

- (1) Il faut d'abord créer une cellule d'adresse P par l'action Allouer(P).
- (2) Une fois cette cellule créée il faut donner la valeur à l'information contenue dans ce nouvel élément de la liste.
- (3) et (4) Enfin, il faut réaliser le chaînage de ce nouvel élément.





```

procédure insérerEnTete(   entrée-sortie Tête : Liste, entrée val : chaîne)
/*
* Insère un nouvel élément dans la liste chaînée passée en paramètre
*/
Variables :
    p : Liste
Début
    allouer(p)                /* Création de la nouvelle cellule dans la liste */
    p^.info ← val
    p^.suivant ← Tête          /* liaison (3) */
    Tête ← p                   /* liaison (4) */
Fin

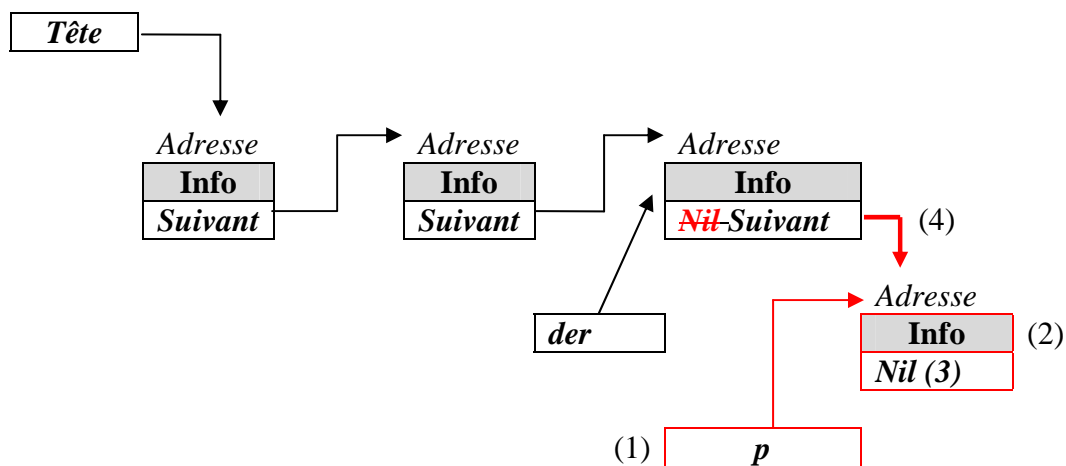
```

## 05-\*\*-Procédure d'insertion en queue de liste chaînée

Si la liste est vide ceci revient à insérer le nouvel élément en tête de liste, sinon :

**Actions à mener :**

- (1) Il faut d'abord créer une cellule d'adresse P par l'action Allouer(P).
- (2) Il faut ensuite donner la valeur au champ Info de ce nouvel élément de la liste.
- (3) et (4) Enfin, il faut réaliser le chaînage de ce nouvel élément au dernier élément de la liste. Pour ce faire il faut connaître l'adresse du dernier élément. Elle sera renvoyée par la fonction *dermier*, que nous écrirons ensuite, et rangée dans la variable *der*.



```

procedure InsérerEnQueue(   entrée-sortie Tête : Liste, entrée val : chaîne)
/* Insère un nouvel élément à la fin de la liste passée en paramètre */
Variables
    p, der : Liste
Début
    si Tête = Nil alors                                     /* La liste est vide */
        insérerEnTete(Tête, val)
    sinon                                                    / la liste n'est pas vide */
        der ← dernier(liste) /* on récupère l'adresse du dernier élément */
1        allouer(p)          /* création de la nouvelle cellule */
2        p^.info ← val        /* stockage de la valeur dans le champ info */
3        p^.suivant ← Nil     /* Liaison (3) : le nouvel élt sera le dernier */
4        der^.suivant ← p     /* Liaison (4) /
    finsi
Fin

```

Pour la fonction *dermier*, qui permet de récupérer l'adresse du dernier élément de la liste, il faut parcourir tous les éléments de la liste en mémorisant à chaque fois l'adresse de la cellule précédente.

```

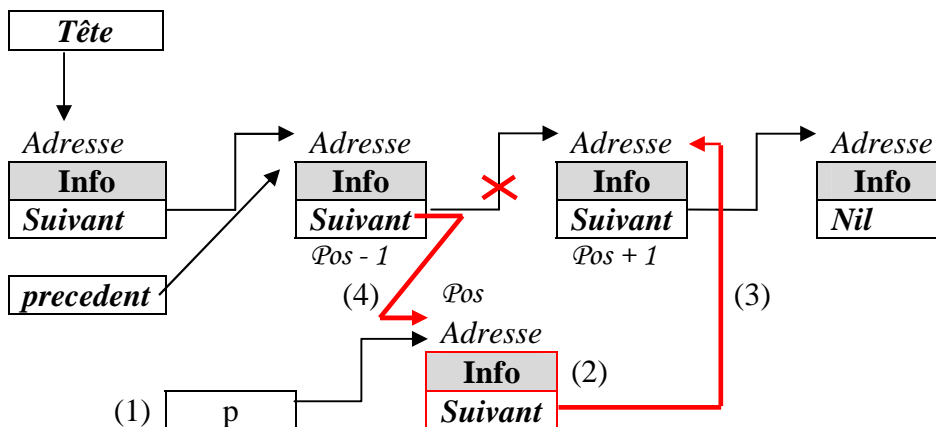
fonction dernier(Tête : Liste) : Liste
    /* Renvoie l'adresse du dernier élément d'une liste chaînée non vide */
Variables
    q : Liste
Début
    q ← Tête
    tantque q^.suivant ≠ Nil
        q ← q^.suivant
    fintantque
    retourner(q)
Fin

```

## 06-\*\*- Procédure d'insertion à une position donnée

Pour insérer à la position désirée, il faut connaître l'adresse de l'élément précédant cette position (position – 1). Si cette position n'existe pas, la procédure ne réalisera pas l'insertion. L'adresse de l'élément qui précédera le nouvel élément est renvoyé par une fonction *accesPosition* qui est écrite ensuite. Cette fonction renvoie Nil si la position n'existe pas.

Il faut prévoir le cas de l'insertion en tête de liste, cas où la position est égale à 1. Dans ce cas il y aura modification de la tête de la liste.



```

procedure insererDansListe( entrée-sortie Tête : Liste,
                           entrée val : chaine, entrée pos : entier)
    /* insertion de la valeur "val" à la position "pos" dans la liste */
Variables
    precedent : Liste
Début
    si pos = 1 alors                                /* on souhaite insérer en tête de liste */
        insererEnTete(Tête, val)
    sinon
        precedent ← accesPosition(Tête, pos - 1) /* voir ci-après */
        si precedent ≠ Nil alors                    /* la position existe */
1           allouer(p)                             /* création de la nouvelle cellule */
2           p^.info ← val
3           p^.suivant ← precedent^.suivant /* Liaison du nouvel élément */
4           precedent^.suivant ← p                /* Liaison de l'élément précédent */
        finsi
    finsi
Fin

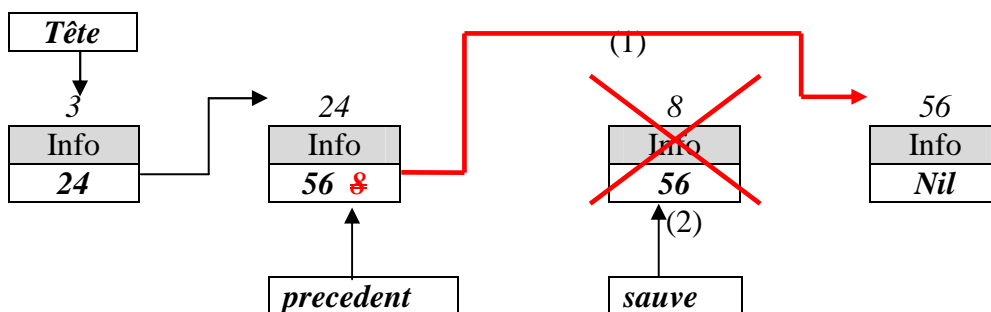
```

```

fonction accesPosition(Tête : Liste, pos : entier) : Liste
    /* Renvoie l'adresse de l'élément qui est en position "pos", ou Nil s'il
    n'y a pas d'élément à cette position. */
Variables
    p : Liste                /* pointeur de parcours */
Début
    p ← Tête
    tantque p ≠ Nil ET pos > 1
        pos ← pos - 1
        p ← p^.suivant
    fintantque
    retourner(p)
Fin

```

## 07-\*\*- Procédure de suppression d'un élément d'une liste chaînée à une position donnée



Comme dans le cas de l'insertion il faut déterminer l'adresse de la cellule qui précède celle à supprimer, c'est-à-dire l'adresse de la position – 1.

```

procedure supprimerDansListe(    entrée-sortie Tête : Liste,
                                entrée pos : entier)
    /* suppression de l'élément à la position "pos" de la liste chaînée */
Variables
    sauve, precedent : Liste
Début
    si Tête ≠ Nil alors                /* liste non vide */
        si pos = 1 alors                /* on souhaite supprimer en tête */
            sauve ← Tête
            Tête ← Tête^.suivant
            desallouer(sauve)
        sinon
            precedent ← accesPosition(Tête, pos - 1)
            si precedent ≠ Nil alors
                sauve ← precedent^.suivant /* adresse élt à supprimer */
                si sauve ≠ Nil alors /* l'élément à supprimer existe */
1                    precedent^.suivant ← sauve^.suivant
2                    desallouer(sauve) /* libère espace de l'élt supprimé */
                finsi
            finsi
        finsi
    finsi
Fin

```

## 08-\*\*- Procédure de calcul de moyenne des étudiants

On a défini les types suivants :

```
Ch10 =   Chaîne de 10 caractères
Ch30 =   Chaîne de 30 caractères
Ent  =   entier
Nb   =   réel
Pe   =   ^Etudiant
Pn   =   ^Note
```

Etudiant =	Structure		Note =	Structure	
	numero	: Ch10		note	: Nb
	nom	: Ch30		coeff	: Ent
	prenom	: Ch30		suivant	: Pn
	moy	: Nb		fin Structure	
	eval	: Pn			
	suivant	: Pe			
	fin Structure				

```
procédure moyennesEtudiants(entrée etu : Pe)
```

```
/* Procédure qui calcule la moyenne de chaque étudiant et met à jour le champ
moy de chaque étudiant de la liste passée en paramètre */
```

```
Variables
```

```
totCoeff : entier
```

```
totNotes : réel
```

```
petu : Pe /* pointeur de parcours de la liste des étudiants */
```

```
pmat : Pn /* pointeur de parcours de la liste des notes de chaque ét.*/
```

```
Début
```

```
petu ← etu
```

```
tantque petu ≠ Nil /* parcours de la liste des étudiants */
```

```
totCoeff ← 0 /* au début il n'y a ni note ni coefficient */
```

```
totNote ← 0 /* pour l'étudiant */
```

```
pmat ← petu^.eval /* eval est la tête de liste des notes de l'et. */
```

```
tantque pmat ≠ Nil /* parcours de la liste des notes de l'étudiant */
```

```
totCoeff ← totCoeff + pmat^.coeff /* somme des coefficients */
```

```
totNote ← totNote + pmat^.note * pmat^.coeff /* somme pondérée */
```

```
pmat ← pmat^.suivant /* on passe à la note suivante */
```

```
fintantque
```

```
si petu^.eval ≠ Nil alors
```

```
/* calcul et mémorisation dans la cellule de l'étudiant de la
moyenne de ses notes, s'il en a */
```

```
petu^.moy ← totNote / totCoeff
```

```
finsi
```

```
fintantque
```

```
Fin
```

## 09-\*\*- Procédure de parcours d'une liste circulaire ou anneau

```
procédure parcoursAnneau(entrée anneau : Liste)
```

```
Variables
```

```
p : Liste
```

```
fini : booléen
```

```
Début /* la liste est supposée non vide */
```

```
P ← anneau
```

```
fini ← faux
```

```
tant que NON(fini)
```

```
traiter(p^.info)
```

```
p ← p^.suivant /* on passe à l'élément suivant de la liste */
```

```
fini ← p = anneau
```

```
fintantque
```

```
Fin
```

## 10-\*\*- Procédure d'insertion d'un élément dans une liste doublement chaînée

On utilise les types suivants :

Type ListeDC = ^Element

Type Element = Structure

    Precedent : ListeDC

    Info : chaîne de caractères

    Suivant : ListeDC

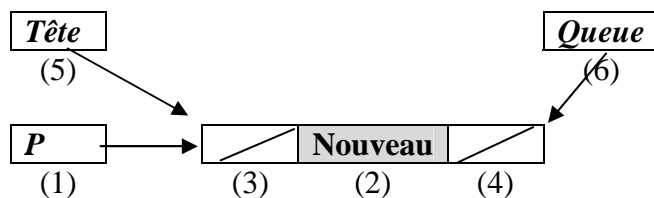
Fin Structure

**Pour insérer un élément dans une liste doublement chaînée il faut :**

- Allouer un emplacement pour le nouvel élément de la liste (1),
- Stocker la valeur à ajouter dans le champ *Info* du nouvel élément (2).

**Si la liste est vide :**

- Donner la valeur *Nil* au champ *Precedent* du nouvel élément (3),
- Donner la valeur *Nil* au champ *Suivant* du nouvel élément (4),
- Donner au pointeur de tête l'adresse du nouvel élément (5),
- Donner au pointeur de queue l'adresse du nouvel élément (6).



```
procedure insererDansListeDoubleVide( entrée-sortie Tête, Queue : ListeDC,
                                     entrée nouveau : chaîne)
```

Variables

    p : ListeDC

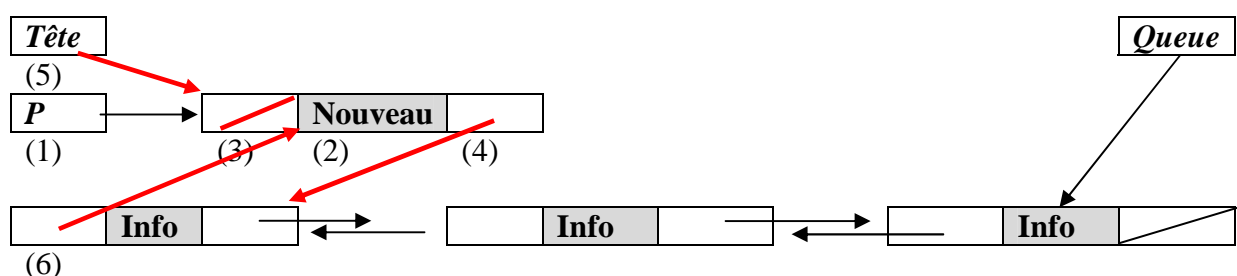
Début

```
(1) allouer(p)                      /* création de la nouvelle cellule dans la liste */
(2) p^.info ← nouveau
(3) p^.precedent ← Nil
(4) p^.suivant ← Nil
(5) Tête ← p
(6) Queue ← p
```

Fin

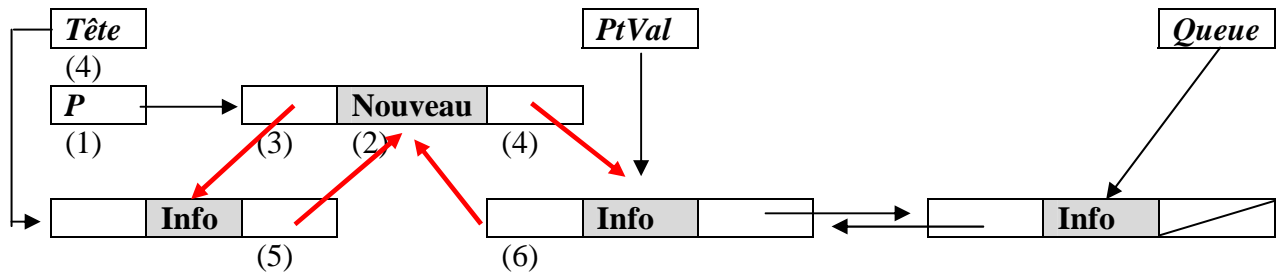
**Si la liste est non vide et que l'insertion se fait en tête :**

- Donner la valeur *Nil* au champ *Precedent* du nouvel élément (3),
- Faire pointer le champ *Suivant* du nouvel élément sur l'élément qui doit le suivre (4),
- Donner au pointeur de tête l'adresse du nouvel élément (5),
- Modifier le pointeur du champ *Precedent* de l'élément suivant qui doit maintenant contenir l'adresse du nouvel élément.



**Si la liste est non vide et que l'insertion se fait avant un élément connu par son adresse :**

- Faire pointer le champ *Precedent* du nouvel élément sur l'élément qui doit le précéder (3),
- Faire pointer le champ *Suivant* du nouvel élément sur l'élément qui doit le suivre (4),
- Modifier le pointeur du champ *Suivant* de l'élément précédent qui doit maintenant contenir l'adresse du nouvel élément (5),
- Modifier le pointeur du champ *Precedent* de l'élément suivant qui doit maintenant contenir l'adresse du nouvel élément (6).



Cette procédure insère l'élément *Nouveau* avant le premier élément portant la valeur "val" dans de la liste doublement chaînée. La fonction *PtV(Tête,Val)* renvoie l'adresse du premier élément qui porte la valeur "val", ou Nil si cette valeur n'existe pas.

```

procedure insererDansListeDouble(entrée-sortie Tête, Queue : ListeDC,
                                entrée val : chaîne,
                                entrée nouveau : chaîne)

    Variables
        p, ptVal : ListeDC
Début
    ptVal ← ptV(Tête, val)
    si ptVal ← Nil alors
(1)      allouer(p) /* création de la nouvelle cellule dans la liste */
(2)      p^.info ← nouveau
(3)      p^.precedent ← ptVal^.precedent
(4)      p^.suivant ← ptVal
        si ptVal = Tête alors /* insertion en tête de liste */
(5)          Tête ← p
        sinon
(5)          ptVal^.precedent^.suivant ← p
        finsi
(6)      ptVal^.precedent ← p
    finsi
Fin

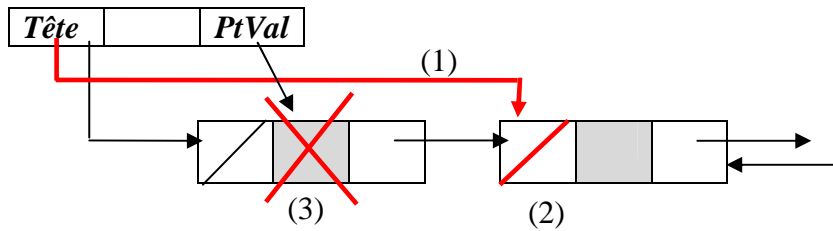
```

## 11-\*\*\*- Procédure de suppression d'un élément dans une liste doublement chaînée

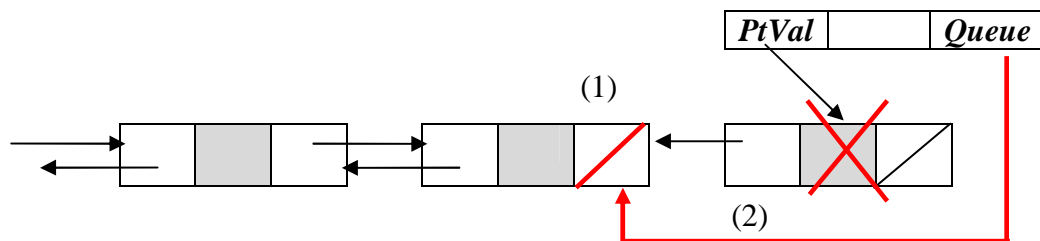
Pour supprimer un élément dans une liste doublement chaînée il faut distinguer 4 cas :

- Suppression du premier élément de la liste ;
- Suppression du dernier élément de la liste ;
- Suppression de l'unique élément de la liste qui est à la fois le premier et le dernier ;
- Suppression d'un élément à une position quelconque de la liste.

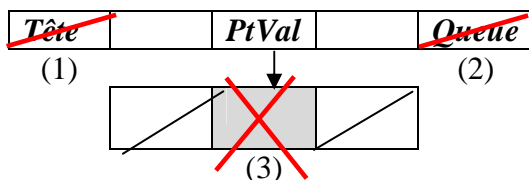
On supprime l'élément pointé par *PtVal*.

*Suppression en tête de liste doublement chaînée :*

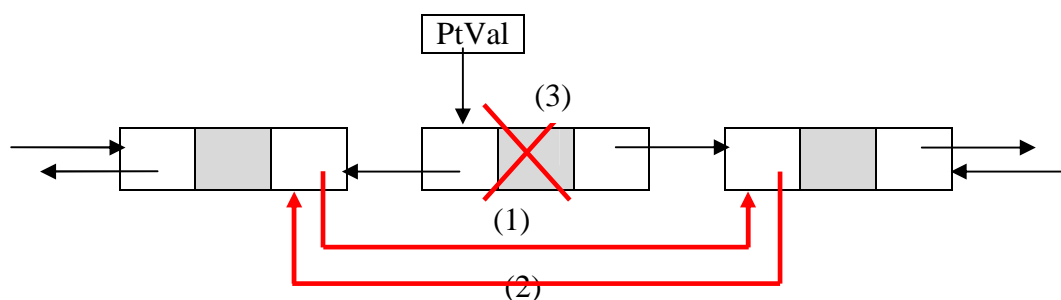
- On fait pointer la tête sur l'élément qui suit l'élément à supprimer (1) ;
- On met Nil dans le pointeur *Precedent* de l'élément qui suit l'élément à supprimer (2);
- On libère l'espace occupé par l'élément à supprimer (3).

*Suppression du dernier élément de la liste doublement chaînée :*

- On fait pointer la queue sur l'élément qui précède l'élément à supprimer (1) ;
- On met Nil dans le pointeur *Suivant* de l'élément qui précède l'élément à supprimer ;
- On libère l'espace occupé par l'élément à supprimer.

*Suppression si la liste doublement chaînée ne contient qu'un seul élément :*

- On met Nil dans le pointeur de tête. (1) ;
- On met Nil dans le pointeur de queue (2) ;
- On libère l'espace occupé par l'élément à supprimer (3).

*Suppression en position quelconque dans la liste doublement chaînée :*

- On modifie le pointeur *Suivant* de l'élément précédant l'élément à supprimer (1) ;
- On modifie le pointeur *Precedent* de l'élément suivant l'élément à supprimer (2) ;
- On libère l'espace occupé par l'élément à supprimer (3).

```

procédure supprimerDansListeDouble(  entrée-sortie Tête, Queue : ListeDC,
                                     entrée val : chaîne,
                                     sortie ok : booléen)

Variables
  ptVal : ListeDC                /* pointeur sur l'élément à supprimer */
Début
  ptVal ← ptV(Tête, val) /* renvoie l'adresse de la première occurrence de
  ok ← faux                "val" ou Nil si celle-ci n'existe pas */
  si ptVal ≠ Nil alors      /* l'elt portant la valeur à supprimer existe */
    ok ← vrai
    si ptVal = Tête alors   /* l'élément est en tête */
(1)      Tête ← Tête^.suivant
    sinon                  /* l'élément n'est pas en tête */
(1)      PtVal^.precedent^.suivant ← ptVal^.suivant
    finsi
    si ptVal = Queue alors /* l'élément est en queue */
(2)      Queue = PtVal^.precedent
    sinon                /* l'élément n'est pas en queue */
(2)      ptVal^.suivant^.precedent ← ptVal^.precedent
    finsi
(3)      desallouer(PtVal)
  finsi
Fin

```

## 12-\*\*\*. Procédure de suppression d'un étudiant dans une liste doublement chaînée

On utilise les types suivants et on travaille avec seulement un pointeur de tête :

```

ListeEtuDC = ^EtudiantDC
EtudiantDC  = Structure
               Precedent : ListeEtuDC
               numero    : chaîne de caractères
               nom       : chaîne de caractères
               prenom    : chaîne de caractères
               moy       : réel
               suivant   : ListeEtuDC
             fin Structure

```

La procédure *supprimerEtudiant* recherche, dans la liste, l'élément dont le numéro est donné en paramètre. Si elle le trouve elle renvoie l'adresse de cet élément et met *vrai* dans *trouvé*. Sinon, elle renvoie *Nil* à la place de l'adresse et *faux* dans *trouvé*. Si l'étudiant a été trouvé, cette procédure appelle la procédure *supprimerDansListeEtuDC* qui réalise effectivement la suppression.

```

procédure supprimerEtudiant(entrée-sortie Tête : ListeEtuDC
                             entrée numEtu : chaîne
                             sortie trouve : booléen)
/* recherche de l'élément portant le numéro de l'étudiant à supprimer et appel
de la procédure de suppression si l'élément est trouvé. */
Variables
  ptEtu : listeEtuDC          /* pointeur de parcours des étudiants */
Début
  trouve ← faux
  ptEtu ← Tête
  tantque ptEtu ≠ Nil et non(trouve)
    si ptEtu^.No = numEtu alors /* on a trouvé l'étududiant */
      trouve ← vrai
      supprimerDansListeEtuDC(Tête, ptEtu)
    sinon
      ptEtu ← ptEtu^.suivant /* on passe à l'étudiant suivant */
    finsi
  fintantque
Fin

```



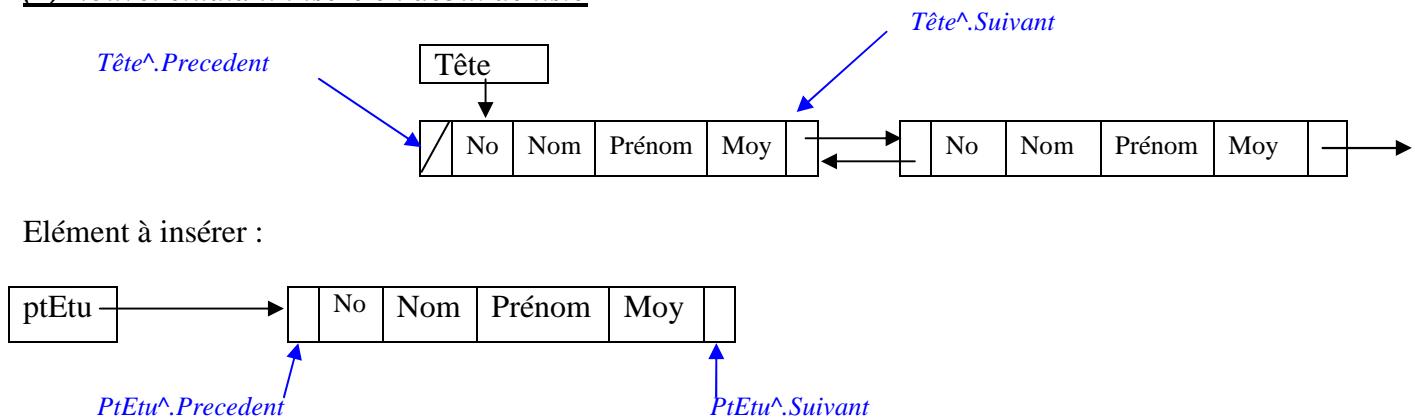
```

procédure supprimerDansListeEtuDC(   entrée-sortie Tête : ListeEtuDC
                                     entrée p : ListeEtuDC)
/* suppression de l'élément pointé par p */
Début
  si p = Tête                        /* l'étudiant à supprimer est en tête de liste */
    Tête ← p^.suivant
  Sinon                               /* mise à jour de l'élément précédant */
    p^.precedent^.suivant ← p^.suivant
  finsi
  si p^.suivant ≠ Nil alors          /* l'étud. à supprimer n'est pas le dernier */
    p^.suivant^.precedent ← p^.precedent
    /* sinon il n'y a pas d'élément suivant à modifier */
  finsi
Fin

```

### 13-\*\*\*- Procédure d'insertion d'un étudiant dans une liste doublement chaînée triée

#### (1) Nouvel étudiant inséré en début de liste

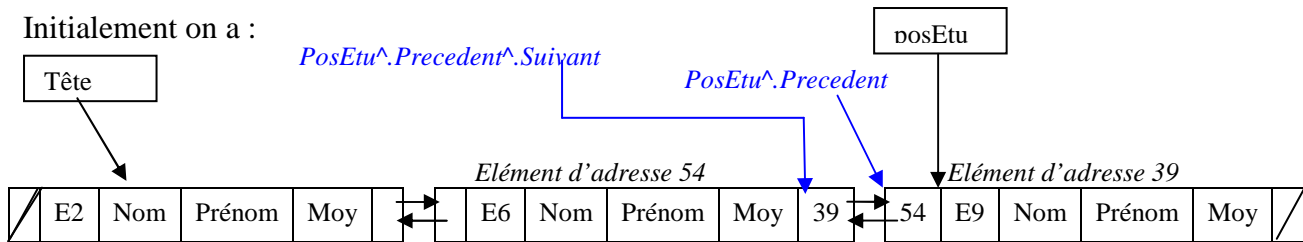


On voit bien sur cette figure que :

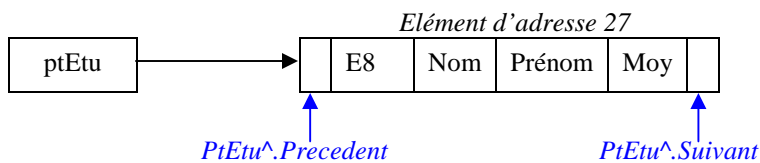
- ptEtu^ qui sera le premier élément de la liste doit avoir :
  - son pointeur ptEtu^.precedent = Nil, puisque c'est le premier il n'y a pas de précédent ;
  - son pointeur ptEtu^.suivant = Tête qui est l'adresse de l'actuel premier élément de la liste qui va devenir le 2<sup>ème</sup> puisque le nouvel étudiant doit être inséré avant lui.
- Tête^ qui sera désormais le 2ème élément de la liste doit avoir :
  - son pointeur Tête^.precedent = ptEtu, puisque l'élément qui va le précéder est celui qui vient d'être créé à l'adresse ptEtu ;
  - son pointeur Tête^.suivant ne change pas. Il peut être égal à Nil ou, comme ici, pointer toujours sur l'élément qui le suivait avant l'insertion du nouvel étudiant;

**(2) Insertion du nouvel élément à la place adéquate dans la liste ici entre le 2<sup>ème</sup> et le 3<sup>ème</sup> élément**

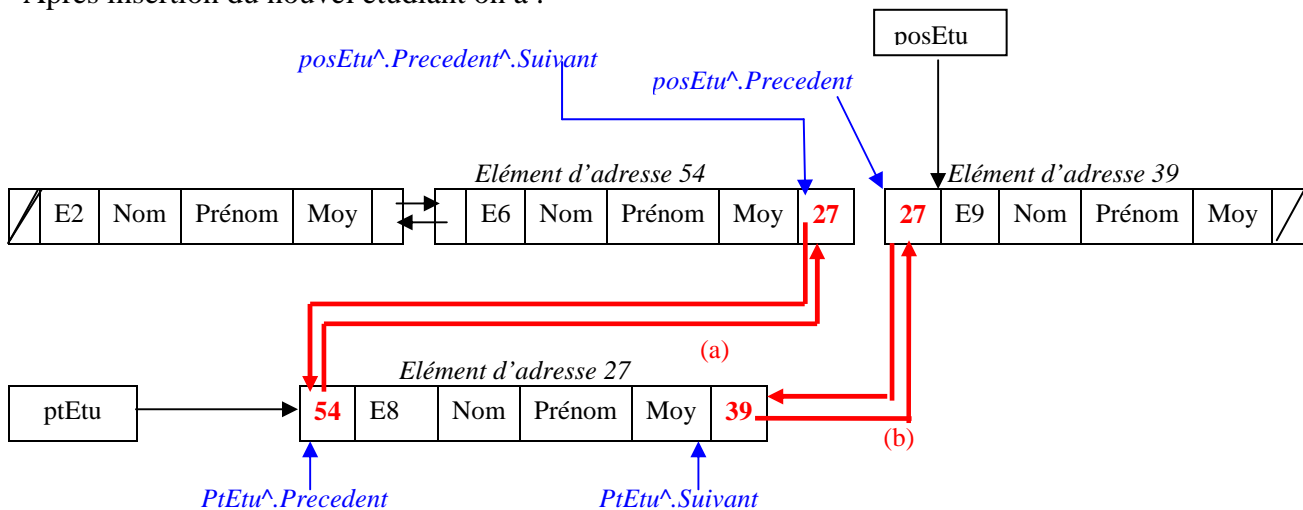
Initialement on a :



Élément à insérer :



Après insertion du nouvel étudiant on a :

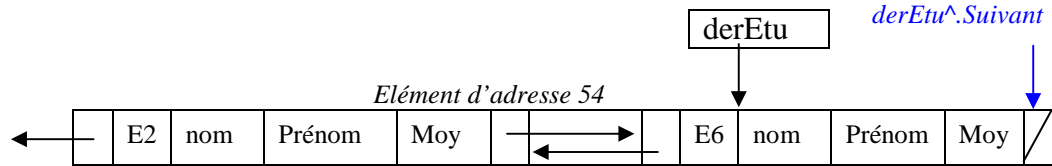


On voit bien sur cette figure que :

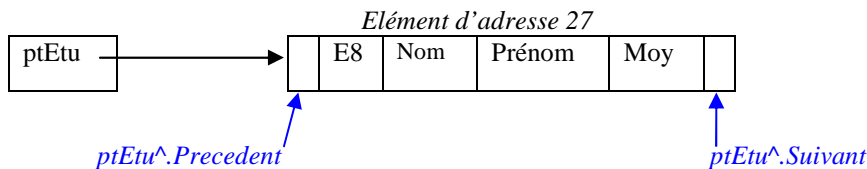
- ptEtu qui sera le 3<sup>ème</sup> élément de la liste doit avoir :
  - son pointeur  $ptEtu.^{precedent} = posEtu.^{precedent}$  (a);
  - son pointeur  $ptEtu.^{suivant} = posEtu$ , adresse de l'élément avant lequel doit se faire l'insertion (b).
- posEtu qui sera désormais le 4<sup>ème</sup> élément de la liste doit avoir :
  - son pointeur  $posEtu.^{precedent} = ptEtu$ , puisque l'élément qui le précède désormais est celui qui vient d'être créé à l'adresse ptEtu ;
- Le 2<sup>ème</sup> élément de la liste doit avoir :
  - son pointeur suivant représenté par  $posEtu.^{precedent}.^{suivant}$  qui doit être égal à ptEtu.

(3) Nouvel étudiant inséré en fin de liste

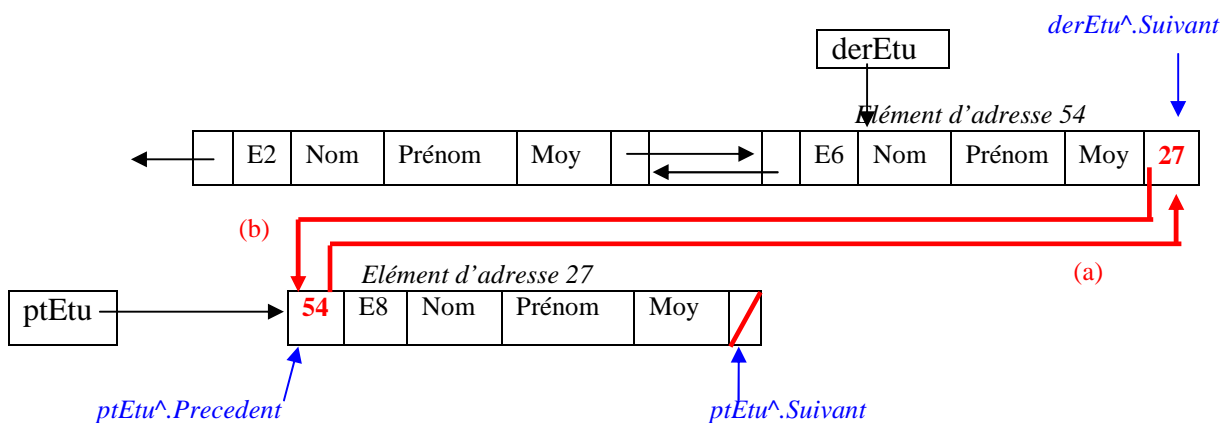
Initialement on a :



Elément à insérer :



Après insertion du nouvel étudiant on a :



On voit bien sur cette figure que :

- ptEtu qui sera le dernier élément de la liste doit avoir :
  - son pointeur ptEtu^.precedent = derEtu (a);
  - son pointeur ptEtu^.suivant = Nil puisqu'il sera le dernier de la liste.
- derEtu^ qui sera désormais l'avant dernier élément de la liste doit avoir :
  - son pointeur derEtu^.suivant = ptEtu, puisque l'élément qui le suit est celui qui vient d'être créé à l'adresse ptEtu (b);

```

procédure insererEtudiant(  entrée-sortie Tête : ListeEtuDC,
                           entrée numEtu, nomEtu, prenomEtu : chaîne)

Variables
  p : ListeEtuDC                /* pointeur de parcours */
  ptEtu : ListeEtuDC
  stop : booléen
Début
  allouer(ptEtu) /* création du nouvel étudiant */
  ptEtu^.No ← numEtu
  ptEtu^.Nom ← nomEtu
  ptEtu^.Prenom ← prenomEtu
  si Tête = Nil alors           /* la liste est vide, l'élément sera unique */
    ptEtu^.suivant ← Nil
    ptEtu^.precedent ← Nil
    Tête ← ptEtu
    Queue ← ptEtu
  sinon
    si numEtu < Tête^.No alors   /* insertion en tête */
      ptEtu^.suivant ← Tête
      ptEtu^.precedent ← Nil
      Tête^.precedent ← ptEtu
      Tête ← ptEtu
    sinon                       /* parcours de la liste jusqu'à trouver le 1er No
      p ← Tête                   supérieur à celui à insérer */
      stop ← faux
      tantque p^.suivant ≠ Nil et non stop faire
        si p^.No > numEtu alors /* insertion avant p^ */
          stop ← vrai
          sinon
            p ← p^.suivant
      fintantque
      si p^.No > numEtu alors /* insertion avant le dernier p^ */
        ptEtu^.suivant ← p
        ptEtu^.precedent ← p^.precedent
        p^.precedent^.suivant ← ptEtu
        p^.precedent ← ptEtu
      sinon                       /* insertion en fin de liste, après p^ */
        ptEtu^.suivant ← Nil
        ptEtu^.precedent ← p
        p^.suivant ← ptEtu
        /* pas d'élément suivant à modifier */
    finsi
  finsi
Fin

```

---

## BIBLIOGRAPHIE

Initiation à l'algorithmique et aux structures de données J. COURTIN et I. KOWARSKI (Dunod)

## WEBGRAPHIE

<http://www.siteduzero.com/tutoriel-3-36245-les-listes-chainees.html>  
[http://liris.cnrs.fr/pierre-antoine.champin/enseignement/algo/listes\\_chainees/](http://liris.cnrs.fr/pierre-antoine.champin/enseignement/algo/listes_chainees/)  
<http://deptinfo.cnam.fr/Enseignement/CycleA/SD/cours/structuress%E9quentielleschain%E9es.pdf>  
<http://pauillac.inria.fr/~maranget/X/421/poly/listes.html#toc2>  
<http://users.skynet.be/Marco.Codutti/esi/web/log2/cours/Liste.pdf>  
<http://membres.lycos.fr/zegour/Publication/Livre3/Cours/Part7.htm>  
<http://wwwens.uqac.ca/~rebaine/8INF805/courslistespilesfiles.pdf>