

# Algorithmique Avancée Langage C Licence 1 SRIT

**KONAN HYACINTHE**

# Chapitre 1 : Pointeurs

## Inconvénients des variables statiques

Les variables « classiques », déclarées avant l'exécution d'un programme, pour ranger les valeurs nécessaires à ce programme, sont des variables statiques, c'est à dire que la place qui leur est réservée en mémoire est figée durant toute l'exécution du programme. Ceci a deux conséquences :

- Risque de manquer de place si la place réservée (par exemple le nombre d'éléments d'un tableau) est trop petite. Il faut alors que le programme prenne en charge le contrôle du débordement.
- Risque de gaspiller de la place si la place réservée est beaucoup plus grande que celle qui est effectivement utilisée par le programme.

Un cahier, dont le nombre de pages est fixé a priori, fournit une bonne image d'une variable statique.

## Introduction des variables dynamiques

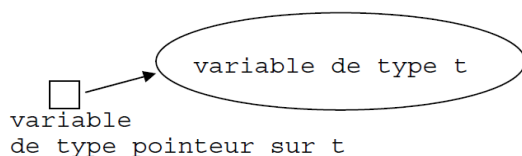
Les variables *dynamiques* vont permettre :

- de prendre la place en mémoire au fur et à mesure des besoins, celle-ci étant bien sûr limitée par la taille de la mémoire,
- de libérer de la place lorsqu'on n'en a plus besoin.

Un classeur dans lequel on peut ajouter ou retirer des pages est une bonne image d'une variable dynamique.

## **Outil : le pointeur**

À partir d'un type **t** quelconque, on peut définir un type **pointeur\_sur\_t**. Les variables du type **pointeur\_sur\_t** contiendront, sous forme d'une adresse mémoire, un mode d'accès au contenu de la variable de type **t**. Celle-ci n'aura pas d'*identificateur*, mais sera accessible uniquement par l'intermédiaire de son *pointeur*.



## Allocation / désallocation d'une zone de mémoire

On a défini un type **t**.


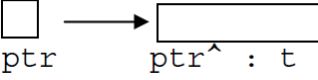
On déclare : **ptr** : **pointeur\_sur\_t** ou **^t**.

### ***Création d'une variable dynamique de type $t$***

#### **ALLOCUER(ptr)**

- réserve un emplacement mémoire de la taille correspondant au type  $t$ ,
- met dans la variable **ptr** l'adresse de la zone mémoire qui a été réservée.


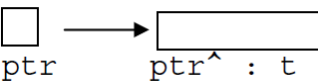
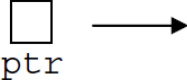
L'emplacement pointé par **ptr** sera accessible par **ptr<sup>^</sup>**.

<b>ptr : <math>^t</math></b>	 ptr
<b>ALLOCUER(ptr)</b>	 ptr → ptr <sup>^</sup> : t

### ***Libération de la place occupée par une variable dynamique***

#### **DESALLOCUER(ptr)**

- libère la place de la zone mémoire dont l'adresse est dans ptr (et la rend disponible pour l'allocation d'autres variables)
- laisse la valeur du pointeur en l'état (n'efface pas l'adresse qui est dans la variable pointeur).

<b>ptr : <math>^t</math></b>	 ptr
<b>ALLOCUER(ptr)</b>	 ptr → ptr <sup>^</sup> : t
<b>DESALLOCUER(ptr)</b>	 ptr →

**Remarque:** si on fait appel au pointeur désalloué, il renvoie une information qui n'a aucun sens.

### **Affectation entre pointeurs**

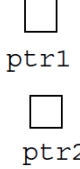
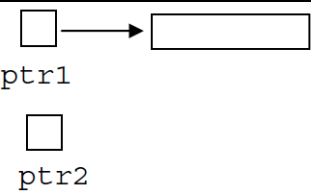
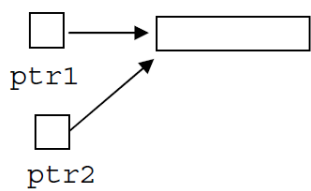
Les pointeurs sont des variables particulières, puisque leurs valeurs sont des adresses mémoires. Ils peuvent néanmoins être impliqués dans des affectations au cours desquelles des adresses sont assignées aux pointeurs.

Pour « vider » un pointeur, c'est à dire annuler l'adresse qu'il contient, on lui affecte une valeur prédéfinie nommée Nil (ou Null). Attention, le fait de mettre Nil dans un pointeur ne libère pas l'emplacement sur lequel il pointait. L'emplacement devient irrécupérable car le lien vers cet emplacement a été coupé par la valeur Nil. Il faut désallouer avant d'affecter le pointeur avec *Nil*.

### Règle de bonne programmation :

Dès qu'on a désalloué un pointeur, il faut lui affecter la valeur Nil, pour qu'il ne conserve pas une adresse mémoire qui n'a plus d'existence physique.

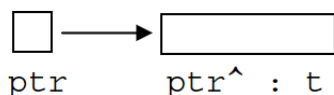
On peut affecter un pointeur avec tout autre pointeur de même type. Après cette affectation, deux pointeurs désignent la même zone de mémoire.

<b>ptr1, ptr2 : pointeur_sur_t</b>	
<b>ALLOUER(ptr1)</b>	
<b>ptr2 ← ptr1</b>	

Pour libérer la zone mémoire on désalloue **ptr1** ou (exclusif) **ptr2**, puis on met la valeur **Nil** dans **ptr1** et dans **ptr2**.

### Accès à la variable pointée

Une fois la variable **ptr** déclarée et allouée, l'accès en écriture ou lecture à la variable pointée par **ptr** se fait avec l'identificateur **ptr^**. On peut appliquer à **ptr^** les mêmes instructions qu'à une variable simple.



Affecter une valeur à la variable pointée : **ptr^ ← <expression de type t>**

lire une valeur de type **t** et la mettre dans la variable pointée : **LIRE(ptr^)**

Afficher la valeur présente dans la zone pointée : **ECRIRE(ptr^)**




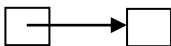


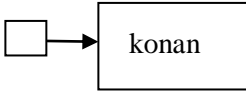


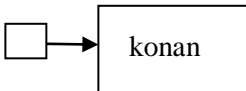
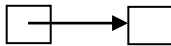

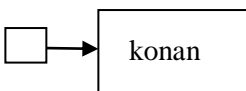
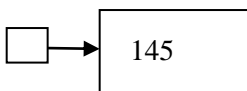

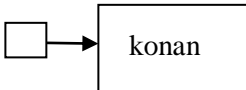
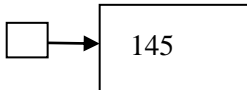

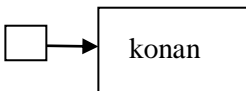
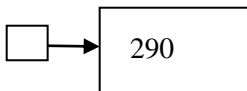

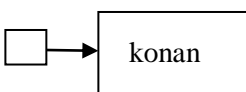
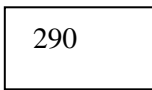

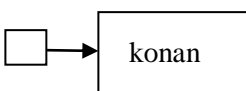


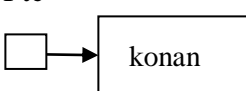





### Un exemple pour comprendre

L'ALGORITHME suivant n'a comme seul but que de faire comprendre la manipulation des pointeurs. À droite les schémas montrent l'état de la mémoire en plus de ce qui s'affiche à l'écran.

### VARIABLES

ptc : pointeur sur **CHAINE**

ptx1, ptx2 : pointeur sur **ENTIER**

<b>VARIABLES</b> ptc : ^CHaine ptx1, ptx2 : ^ENTIER	Ptc 	Ptx1 	Ptx2 
<b>ALLOUER</b> (ptc)	Ptc 	Ptx1 	Ptx2 
ptc^ ← "konan"	Ptc 	Ptx1 	Ptx2 
<b>ECRIRE</b> (ptc^)	Konan		
<b>ALLOUER</b> (ptx1)	Ptc 	Ptx1 	Ptx2 
<b>LIRE</b> (ptx1^)	Ptc 	Ptx1 	Ptx2 
Ptx2 ← Ptx1	Ptc 	Ptx1 	Ptx2 
<b>ECRIRE</b> (ptx2^)	145		
ptx2^ ← ptx1^ + ptx2^	Ptc 	Ptx1 	Ptx2 
<b>ECRIRE</b> (ptx1^, ptx2^)	290 290		
ptx1 ← NIL	Ptc  Ptc	Ptx1 	Ptx2 
<b>ECRIRE</b> (ptx2^)	290		
<b>DESALLOUER</b> (ptx2)	Ptc 	Ptx1 	Ptx2 
Ptx2 ← NIL	Ptc 	Ptx1 	Ptx2 
ptc ← NIL	Ptc 	Ptx1 	Ptx2 

**Où est l'erreur ?** : on a coupé l'accès à la zone de mémoire sans la désallouer. Elle est irrécupérable

# Chapitre 2 : Listes chaînées

## 1. Structures de données linéaires

Parmi les structures de données linéaires il y a :

- les tableaux,
- les Listes chaînées,
- les piles,
- les files.

Les structures de données linéaires induisent une notion de séquence entre les éléments les composant (1er, 2ème, 3ème, suivant, dernier...).

### 1.1. Les tableaux

Vous connaissez déjà la structure linéaire de type tableau pour lequel les éléments de même type le composant sont placés de façon contigüe en mémoire.

Pour créer un tableau, à 1 ou 2 dimensions, il faut connaître sa taille qui ne pourra être modifiée au cours du programme, et lui associer un indice pour parcourir ses éléments. Pour les tableaux la séquence correspond aux numéros des cases du tableau. On accède à un élément du tableau directement grâce à son indice.

Soit le tableau à 1 dimension suivant nommé **Tablo** :

12	14	10	24	
----	----	----	----	--

pour atteindre la troisième case du tableau il suffit d'écrire **Tablo[3]** qui contient 10, si les valeurs de l'indice commencent à 1.

La structure de type tableau pose des problèmes pour insérer ou supprimer un élément car ces actions nécessitent des décalages du contenu des cases du tableau qui prennent du temps dans l'exécution d'un programme.

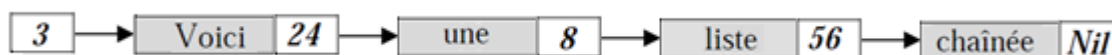
Ce type de stockage de valeurs peut donc être coûteux en temps d'exécution. Il existe une autre structure, appelée Liste chaînée, pour stocker des valeurs, cette structure permet plus aisément d'insérer et de supprimer des valeurs dans une Liste linéaire d'éléments.

### 1.2. Les Listes chaînées

Une **Liste chaînée** est une structure linéaire qui n'a pas de dimension fixée à sa création. Ses éléments de même type sont éparpillés dans la mémoire et reliés entre eux par des pointeurs. Sa dimension peut être modifiée selon la place disponible en mémoire. La Liste est accessible uniquement par sa tête de Liste c'est-à-dire son premier élément.

Pour les Listes chaînées la séquence est mise en œuvre par le pointeur porté par chaque élément qui indique l'emplacement de l'élément suivant. Le dernier élément de la Liste ne pointe sur rien (Nil). On accède à un élément de la Liste en parcourant les éléments grâce à leurs pointeurs.

Soit la **Liste** chaînée suivante (@ indique que le nombre qui le suit représente une adresse) :



Pour accéder au troisième élément de la Liste il faut toujours débiter la lecture de la Liste par son premier élément dans le pointeur duquel est indiqué la position du deuxième élément. Dans le pointeur du deuxième élément de la Liste on trouve la position du troisième élément...

Pour ajouter, supprimer ou déplacer un élément il suffit d'allouer une place en mémoire et de mettre à jour les pointeurs des éléments.

Il existe différents types de Listes chaînées :

- **Liste chaînée simple** constituée d'éléments reliés entre eux par des pointeurs.
- **Liste chaînée ordonnée** où l'élément suivant est plus grand que le précédent. L'insertion et la suppression d'élément se font de façon à ce que la **Liste** reste triée.
- **Liste doublement chaînée** où chaque élément dispose non plus d'un mais de deux pointeurs pointant respectivement sur l'élément précédent et l'élément suivant. Ceci permet de **lire** la **Liste** dans les deux sens, du premier vers le dernier élément ou inversement.
- **Liste circulaire** où le dernier élément pointe sur le premier élément de la **Liste**. S'il s'agit d'une **Liste** doublement chaînée alors le premier élément pointe également sur le dernier.

Ces différents **TYPE** s peuvent être mixés selon les besoins.

On utilise une **Liste** chaînée plutôt qu'un tableau lorsque l'on doit traiter des objets représentés par des suites sur lesquelles on doit effectuer de nombreuses suppressions et de nombreux ajouts. Les manipulations sont alors plus rapides qu'avec des tableaux.

*Résumé*

Structure	Dimension	Position d'une information	Accès à une information
<b>Tableau</b>	Fixe	Par son indice	Directement par l'indice
<b>Liste chaînée</b>	Evolue selon les actions	Par son adresse	Séquentiellement par le pointeur de chaque élément

### 1.3. Les piles et les files

Les files et les piles sont des Listes chaînées particulières qui permettent l'ajout et la suppression d'éléments uniquement à une des deux extrémités de la Liste.

Structures	Ajout	Suppression	TYPE de Liste
<b>PILE</b>	Tête	Tête	LIFO (Last In First Out)
<b>FILE</b>	Queue	Tête	FIFO (First In First Out)

La pile est une structure de **Liste** similaire à une pile d'assiettes où l'on pose et l'on prend au sommet de la pile.

La file est une structure de **Liste** similaire à une file d'attente à une caisse, le premier client entré dans la file est le premier sorti de celle-ci (aucun resquillage n'est admis).

## 2. Listes chaînées

### 2.1. Définitions

Un **élément** d'une **Liste** est l'ensemble (ou structure) formé :

- d'une donnée ou information,
- d'un pointeur nommé *Suivant* indiquant la position de l'élément suivant dans la **Liste**.

A chaque élément est associée une adresse mémoire.

Les **Listes** chaînées font appel à la notion de variable dynamique.

Une variable dynamique:

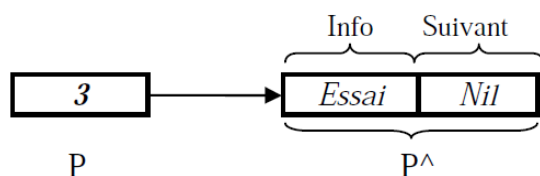
- est déclarée au début de l'exécution d'un programme,
- elle y est créée, c'est-à-dire qu'on lui alloue un espace à occuper à une adresse de la mémoire,
- elle peut y être détruite, c'est-à-dire que l'espace mémoire qu'elle occupait est libéré,
- l'accès à la valeur se fait à l'aide d'un pointeur.

Un **POINTEUR** est une variable dont la valeur est une adresse mémoire (voir chapitre 1). Un pointeur, noté **P**, pointe sur une variable dynamique notée **P<sup>^</sup>**.

Le **type de base** est le **type** de la variable pointée.

Le **type du pointeur** est l'ensemble des adresses des **variables** pointées du **type** de base. Il est représenté par le symbole <sup>^</sup> suivi de l'identificateur du **type** de base.

*Exemple:*



La variable pointeur **P** pointe sur l'espace mémoire **P<sup>^</sup>** d'adresse 3. Cette cellule mémoire contient la valeur "Essai" dans le champ *Info* et la valeur spéciale *Nil* dans le champ *Suivant*. Ce champ servira à indiquer quel est l'élément suivant lorsque la cellule fera partie d'une **Liste**. La valeur *Nil* indique qu'il n'y a pas d'élément suivant. **P<sup>^</sup>** est l'objet dont l'adresse est rangée dans **P**.

Les **Listes** chaînées entraînent l'utilisation de procédures d'allocation et de libération dynamiques de la mémoire. Ces procédures sont les suivantes :

- **ALLOCUER(P)** : réserve un espace mémoire **P<sup>^</sup>** et donne pour valeur à **P** l'adresse de cet espace mémoire. On alloue un espace mémoire pour un élément sur lequel pointe **P**.
- **DESALLOCUER(P)** : libère l'espace mémoire qui était occupé par l'élément à supprimer **P<sup>^</sup>** sur lequel pointe **P**.



Pour définir les **variables** utilisées dans l'exemple ci-dessus, il faut :

- définir le type des éléments de **Liste** :

**TYPE** Cellule = **STRUCTURE**

Info : **CHAINE**

Suivant : **Liste**

**FINSTRUCTURE**

- définir le type du pointeur : **TYPE Liste** = ^Cellule
- déclarer une variable pointeur : **VARIABLES P** : **Liste**
- **ALLOUER** une cellule mémoire qui réserve un espace en mémoire et donne à P la valeur de l'adresse de l'espace mémoire P^ : **ALLOUER(P)**
- affecter des valeurs à l'espace mémoire P^ : P^.Info ← "Essai" ; P^.Suivant ← Nil

Quand P = Nil alors P ne pointe sur rien.

## 2.2. Listes chaînées simples

Une **Liste** chaînée simple est composée :

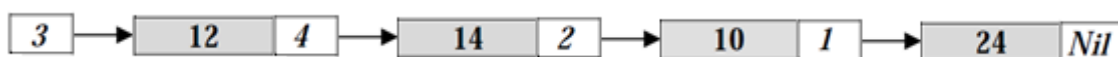
- d'un ensemble d'éléments tel que chacun :
  - est rangé en mémoire à une certaine adresse,
  - contient une donnée (*Info*),
  - contient un pointeur, souvent nommé *Suivant*, qui contient l'adresse de l'élément suivant dans la **Liste**,
- d'une variable, appelée *Tête*, contenant l'adresse du premier élément de la **Liste** chaînée.

Le pointeur du dernier élément contient la valeur *Nil*. Dans le cas d'une **Liste** vide le pointeur de la tête contient la valeur *Nil*. Une **Liste** est définie par l'adresse de son premier élément.

Avant d'écrire des algorithmes manipulant une Liste chaînée, il est utile de montrer un schéma représentant graphiquement l'organisation des éléments de la Liste chaînée.

*Exemple:*

Reprenons l'exemple du tableau paragraphe 1.1. La **Liste** chaînée correspondante pourrait être :



- Le 1er élément de la **Liste** vaut 12 à l'adresse 3 (début de la **Liste** chaînée)
- Le 2e élément de la **Liste** vaut 14 à l'adresse 4 (car le pointeur de la cellule d'adresse 3 est égal à 4)
- Le 3e élément de la **Liste** vaut 10 à l'adresse 2 (car le pointeur de la cellule d'adresse 4 est égal à 2)
- Le 4e élément de la **Liste** vaut 24 à l'adresse 1 (car le pointeur de la cellule d'adresse 2 est égal à 1)

*Si P a pour valeur 3*

**P^.Info** a pour valeur 12

**P^.Suivant** a pour valeur 4

*Si P a pour valeur 2*

**P^.Info** a pour valeur 10

**P^.Suivant** a pour valeur 1

### **2.3. Traitements de base d'utilisation d'une Liste chaînée simple**

Il faut commencer par définir un type de variable pour chaque élément de la chaîne. En langage algorithmique ceci se fait comme suit :

**TYPE Liste = ^Element**

**TYPE Element = STRUCTURE**

Info : **VARIANT**

Suivant : **Liste**

**FINSTRUCTURE**

**VARIABLES Tete, P : Liste**

Le type de **Info** dépend des valeurs contenues dans la **Liste** : **ENTIER, CHAINE, VARIANT** pour un type quelconque.

Les traitements des **Listes** sont les suivants :

- Créer une **Liste**.
- Ajouter un élément.
- Supprimer un élément.
- Modifier un élément.
- Parcourir une **Liste**.
- Rechercher une valeur dans une **Liste**.

#### **2.3.1 Créer une Liste chaînée composée de 2 éléments de TYPE chaîne de caractères**

**Déclarations des types pour la Liste :**

**TYPE Liste = ^Element**

**TYPE Element = STRUCTURE**

Info : **CHAINE**

Suivant : **Liste**

**FINSTRUCTURE**

## ALGORITHME CréationListe2Elements

### VARIABLES

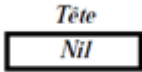
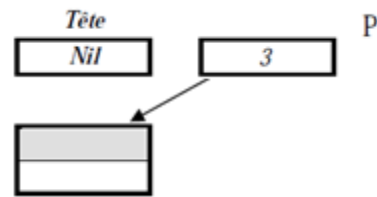
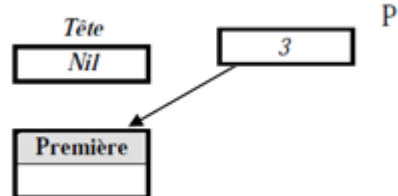
Tete, P : **Liste**

NombreElt : **ENTIER**

### DEBUT

```
1  Tete ← Nil /*pour l'instant la liste est vide*/
2  ALLOUER(P) /* réserve un espace mémoire pour le premier élément */
3  LIRE(P^.Info) /* stocke dans l'Info de l'élément pointé par P la valeur saisie */
4  P^.Suivant ← Nil /* il n'y a pas d'élément suivant */
5  Tete ← P /* le pointeur Tete pointe maintenant sur P */
/* Il faut maintenant ajouter le 2e élément, ce qui revient à insérer un élément en tête de liste */
6  ALLOUER(P) /* réserve un espace mémoire pour le second élément */
7  LIRE(P^.Info) /* stocke dans l'Info de l'élément pointé par P la valeur saisie */
8  P^.Suivant ← Tete /* élément inséré en tête de liste */
9  Tete ← P
```

### FIN

1	Tete ← Nil	 <p>A box labeled 'Tete' with 'Nil' inside it.</p>
2	ALLOUER(P)	 <p>A box labeled 'Tete' with 'Nil' inside it. To its right is a box labeled '3' with 'P' to its right. An arrow points from the '3' box to a new empty box below 'Tete'.</p>
3	LIRE(P^.Info)	 <p>A box labeled 'Tete' with 'Nil' inside it. To its right is a box labeled '3' with 'P' to its right. An arrow points from the '3' box to a box labeled 'Première' below 'Tete'.</p>

4	$P^{\wedge}.Suivant \leftarrow Nil$	<p>Diagram for step 4: A pointer <math>P</math> points to a node containing the value 3. The node's next pointer (Suivant) points to the 'Première' pointer, which currently points to Nil.</p>
5	$Tete \leftarrow P$	<p>Diagram for step 5: The 'Tête' pointer now points to the node containing the value 3. <math>P</math> still points to the same node.</p>
6	<b>ALLOUER</b> (P)	<p>Diagram for step 6: A new node is allocated and pointed to by <math>P</math>. The node is empty.</p>
7	<b>LIRE</b> ( $P^{\wedge}.Info$ )	<p>Diagram for step 7: The new node now contains the value 24. <math>P</math> points to it.</p>
8	$P^{\wedge}.Suivant \leftarrow Tete$	<p>Diagram for step 8: The next pointer of the new node (24) is set to <math>Tete</math>, which points to the first node (3).</p>
9	$Tete \leftarrow P$	<p>Diagram for step 9: The 'Tête' pointer is updated to point to the new node (24).</p>

## Traduction en C(Liste d'ENTIERs)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Typedef struct Element
{
    int Info;
    struct Element* Suivant;
}Element;

Typedef Element* Liste;
Liste Tete, p ;
char input[10];

int main()
{
    Tete = NULL ;

    (p) = malloc(sizeof(Element));
    printf("SAISISSEZ UN ENTIER :\n");
    scanf("%s", input);
    p->Info = atoi(input);
    p->Suivant = NULL ;
    Tete = p ;

    (p) = malloc(sizeof(Element));
    printf("SAISISSEZ UN ENTIER :\n");
    scanf("%s", input);
    p->Info = atoi(input);
    p->Suivant = Tete;
    Tete = p ;

    printf("LA LISTE SAISIE EST :\n");
    printf("\n");
    while (p != NULL)
    {
        printf("-> %i ", p->Info);
        p = p->Suivant;
    }
    printf("TERMINER :\n");
} OK
```

## Traduction en C (Liste de chaîne de caractères)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Element
{
    char Info[10];
    struct Element* Suivant;
}Element;

typedef Element*Liste;
Liste Tete, p ;
char input[10];

int main()
{
    Tete = NULL ;

    (p) = malloc(sizeof(Element));
    printf("SAISISSEZ UNE CHAINE :\n");
    scanf("%s", input);
    strcpy (p->Info, input);
    p->Suivant = NULL ;
    Tete = p ;

    (p) = malloc(sizeof(Element));
    printf("SAISISSEZ UNE CHAINE :\n");
    scanf("%s", input);
    strcpy (p->Info, input);
    p->Suivant = Tete;
    Tete = p ;

    printf("LA LISTE SAISIE EST :\n");
    printf("\n");
    while (p != NULL)
    {
        printf("->%s ", p->Info);
        p = p->Suivant;
    }
    printf("TERMINER :\n");
} OK
```

### 2.3.2 Créer une Liste chaînée composée de plusieurs éléments de type chaîne de caractères

Déclarations des types pour la Liste :

```
TYPE Liste = ^Element
TYPE Element = STRUCTURE
    Info : CHAINE
    Suivant : Liste
FINSTRUCTURE
```

Pour créer une Liste chaînée contenant un nombre d'éléments à préciser par l'utilisateur il suffit d'introduire deux variables de type ENTIER NombreElt et Compteur

- de faire saisir la valeur de *NombreElt* par l'utilisateur dès le début du programme,
- d'écrire une boucle pour *Compteur allant de 1 à NombreElt* comprenant les instructions 6, 7, 8 et 9.

#### ALGORITHME CréationListeNombreConnu

##### VARIABLES

```
Tete, P : Liste
NombreElt : ENTIER
Compteur : ENTIER
```

##### DEBUT

```
LIRE(NombreElt)
```

```
Tete ← Nil
```

```
POUR Compteur DE 1 A NombreElt FAIRE
```

```
6     ALLOUER(P) /* réserve un espace mémoire pour l'élément à ajouter */
```

```
7     LIRE(P^.Info) /* stocke dans l'Info de l'élément pointé par P la valeur saisie */
```

```
8     P^.Suivant ← Tete /* élément inséré en tête de Liste */
```

```
9     Tete ← P /* le pointeur Tete pointe maintenant sur P */
```

```
FINPOUR
```

##### FIN

## Traduction en C(Liste d'entiers)

```
#include <stdio.h>
#include <stdlib.h>

Typedef struct Element
{
    int Info;
    struct Element* Suivant;
}Element;

TypedefElement*Liste;
Liste Tete, p ;
int NombreElt ;
int Compteur ;

char input[10];

int main()
{
    printf("ENTREZ LE NOMBRE D'ELEMENT DE LA LISTE : ");
    scanf("%d",&NombreElt);
    Tete = NULL ;
    for (Compteur = 0; Compteur < NombreElt; Compteur ++ )
    {
        (p) = malloc(sizeof(Element));
        printf("SAISISSEZ UN ENTIER :\n");
        scanf("%s", input);
        p->Info = atoi(input);
        p->Suivant = Tete;
        Tete = p ;
    }
    printf("LA LISTE SAISIE EST :\n");
    printf("\n");
    while (p != NULL)
    {
        printf("-> %i", p->Info);
        p = p->Suivant;
    }
    printf("TERMINER :\n");
} OK
```



## Traduction en C (Liste de chaîne de caractères)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Typedef struct Element
{
    int Info[10];
    struct Element* Suivant;
}Element;

TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];
int NombreElt, Compteur;

int main()
{
    Tete = NULL ;
    printf("ENTREZ LE NOMBRE D'ELEMENT DE LA LISTE : ");
    scanf("%d",&NombreElt);

    for (Compteur = 0; Compteur < NombreElt; Compteur ++ )
    {
        (p) = malloc(sizeof(Element));
        printf("SAISISSEZ UNE CHAINE :\n");
        scanf("%s", Valeur);
        strcpy (p->Info, Valeur);
        p->Suivant = Tete;
        Tete = p ;
    }
    printf("LA LISTE SAISIE EST :\n");
    printf("\n");
    while (p != NULL)
    {
        printf("->%s ", p->Info);
        p = p->Suivant;
    }
    printf("TERMINER :\n");
} OK
```

pour créer une Liste chaînée contenant un nombre indéterminé d'éléments il faut :

- déclarer une variable de lecture de même type que celui de l'information portée par la Liste,
- déterminer et indiquer à l'utilisateur la valeur qu'il doit saisir pour annoncer qu'il n'y a plus d'autre élément à ajouter dans la chaîne (ici "X"),
- écrire une boucle Tant Que permettant d'exécuter les instructions 6, 7, 8 et 9 tant que la valeur saisie par l'utilisateur est différente de la valeur indiquant la fin de l'ajout d'élément dans la chaîne.

#### **ALGORITHME CréationListeNombreInconnu**

##### **VARIABLES**

Tete, P : Liste

Valeur : CHAÎNE

##### **DEBUT**

Tete ← Nil

**LIRE**(Valeur)

**TANTQUE** Valeur ≠ "X" **FAIRE**

**ALLOUER**(P) /\* réserve un espace mémoire pour l'élément à ajouter \*/

    P^.Info ← Valeur /\* stocke dans l'Info de l'élément pointé par P la valeur saisie \*/

    P^.Suivant ← Tete /\* élément inséré en tête de Liste \*/

    Tete ← P /\* le pointeur Tete pointe maintenant sur P \*/

**LIRE**(Valeur)

**FINTANTQUE**

##### **FIN**

## Traduction en C (Liste d'entiers)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Typedef struct Element
{
    int Info;
    struct Element* Suivant;
}Element;

TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];

int main()
{
    Tete = NULL ;
    printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
    scanf("%s", Valeur);
    while (atoi(Valeur) != -1)
    {
        (p) = malloc(sizeof(Element));
        p->Info = atoi(Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
        scanf("%s", Valeur);
    }
    printf("LA LISTE SAISIE EST :\n");
    printf("\n");
    while (p != NULL)
    {
        printf("-> %i ", p->Info);
        p = p->Suivant;
    }
    printf("TERMINER :\n");
} OK
```

## Traduction en C (Liste de chaîne de caractères)

```
#include <stdio.h>
#include <stdlib.h>

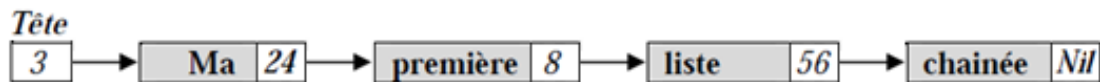
Typedef struct Element
{
    char Info[10];
    struct Element* Suivant;
}Element;

TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];

int main()
{
    Tete = NULL ;
    printf("SAISIR UNE CHAÎNE ('x' POUR FINIR) : ");
    scanf("%s", Valeur);
    while (Valeur[0] != 'x')
    {
        (p) = malloc(sizeof(Element));
        strcpy(p->Info, Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UNE CHAÎNE ('x' POUR FINIR) : ");
        scanf("%s", Valeur);
    }
    printf("LA LISTE SAISIE EST :\n");
    printf("\n");
    while (p != NULL)
    {
        printf("->%s ", p->Info);
        p = p->Suivant;
    }
    printf("TERMINER :\n");
} OK
```

### 2.3.3. Afficher les éléments d'une Liste chaînée

Une **Liste** chaînée simple ne peut être parcourue que du premier vers le dernier élément de la **Liste**.



L'ALGORITHME est donné sous forme d'une procédure qui reçoit la tête de Liste en paramètre.

#### **PROCEDURE AfficherListe( (E) P : Liste)**

*/\* Afficher les éléments d'une **Liste** chaînée passée en paramètre \*/*

#### **DEBUT**

**1** P ← Tete */\* P pointe sur le premier élément de la **Liste**\*/*

*/\* On parcourt la **Liste** tant que l'adresse de l'élément suivant n'est pas Nil \*/*

**2 TANTQUE P <> NIL FAIRE***/\* si la **Liste** est vide Tete est à Nil \*/*

**2.1 ECRIRE**(P^.Info) */\* afficher la valeur contenue à l'adresse pointée par P \*/*

**2.2** P ← P^.Suivant */\* On passe à l'élément suivant \*/*

**FINTANTQUE**

#### **FIN**

1	P a pour valeur 3
2	"Ma" s'affiche
3	P prend pour valeur 24
2	"première" s'affiche
3	P prend pour valeur 8
2	"Liste" s'affiche
3	P prend pour valeur 56
2	"chaînée" s'affiche
3	P prend pour valeur Nil

On s'arrête puisque P a pour valeur Nil et que c'est la condition d'arrêt de la boucle **TantQue**.

## Traduction en C (Liste d'ENTIERs)

```
#include <stdio.h>
#include <stdlib.h>

Typedef struct Element
{
    int Info;
    struct Element* Suivant;
}Element;

TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];

/*****
* Affiche le contenu de la Liste L *
*****/
void AfficherListe(Liste L)
{
    Liste tmp = L;
    while (tmp != NULL)
    {
        printf("->%d",tmp->Info);
        tmp = tmp->Suivant;
    }
    printf("->NULL\n");
}

int main()
{
    Tete = NULL ;
    printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
    scanf("%s", Valeur);
    while (atoi(Valeur) != -1)
    {
        (p) = malloc(sizeof(Element));
        p->Info = atoi(Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
        scanf("%s", Valeur);
    }
    printf("\n");
    printf(" AFFICHAGE DE LA LISTE : \n");
    printf("\n");
    AfficherListe(Tete);
    return 0;
} OK
```

## Traduction en C (Liste de chaine de caractères)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Typedef struct Element
{
    char Info[10];
    struct Element* Suivant;
}Element;

TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];

/*****
* Affiche le contenu de la Liste L *
*****/
void AfficherListe(Liste L)
{
    Liste tmp = L;
    while (tmp != NULL)
    {
        printf("->%s", tmp->Info);
        tmp = tmp->Suivant;
    }
    printf("->NULL\n");
}

int main()
{
    Tete = NULL ;
    printf("SAISIR UNE CHAINE ('x' POUR FINIR) : ");
    scanf("%s", Valeur);
    while (Valeur[0] != 'x')
    {
        (p) = malloc(sizeof(Element));
        strcpy(p->Info, Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UNE VHAINE ('x' POUR FINIR) : ");
        scanf("%s", Valeur);
    }
    printf("\n");
    printf(" AFFICHAGE DE LA LISTE : \n");
    printf("\n");
    AfficherListe(Tete);
    return 0;
} OK
```

### 2.3.4. Rechercher une valeur donnée dans une Liste chaînée ordonnée

Dans cet exemple nous reprenons le cas de la **Liste** chaînée contenant des éléments de type chaîne de caractères, mais ce pourrait être tout autre type, selon celui déterminé à la création de la **Liste** (rappelons que tous les éléments d'une **Liste** chaînée doivent avoir le même type). La **Liste** va être parcourue à partir de son premier élément (celui pointé par le pointeur de tête). Il a deux cas d'arrêt :

- avoir trouvé la valeur de l'élément,
- avoir atteint la fin de la **Liste**.

L'ALGORITHME est donné sous forme d'une procédure qui reçoit la tête de Liste en paramètre. La valeur à chercher est lue dans la procédure.

**PROCEDURE RechercherValeurListe**((E) Tete : **Liste**, (E) Val : **chaîne**)

/\* Rechercher si une valeur donnée en paramètre est présente dans la **Liste** passée en paramètre \*/

**VARIABLES**

P : **Liste** /\* pointeur de parcours de la **Liste**\*/

Trouve : booléen /\* indicateur de succès de la recherche \*/

**DEBUT**

**SI** Tete <> Nil **ALORS**/\* la **Liste** n'est pas vide on peut donc y chercher une valeur \*/

P ← Tete

Trouve ← Faux

**TANTQUE** P <> Nil **ET Non** Trouve **FAIRE**

**SI** P^.Info = Val **ALORS**/\* L'élément recherché est l'élément courant \*/

Trouve ← Vrai

**SINON**/\* L'élément courant n'est pas l'élément recherché \*/

P ← P^.Suivant /\* on passe à l'élément suivant dans la **Liste**\*/

**FINSI**

**FINTANTQUE**

**SI** Trouve **ALORS**

**ECRIRE**(" La valeur ", Val, " est dans la **Liste**")

**SINON**

**ECRIRE**(" La valeur ", Val, " n'est pas dans la **Liste**")

**FINSI**

**SINON**

**ECRIRE**("La **Liste** est vide")

**FINSI**

**FIN**



## Traduction en C (Liste d'ENTIERs)

```
#include <stdio.h>
#include <stdlib.h>
```

```
Typedef struct Element
{
    int Info;
    struct Element* Suivant;
}Element;
```

```
TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];
```

```
/******
* Recherche la valeur val dans la Liste L *
******/
void RechercherValeurListe(Liste L,int val)
{
    Liste tmp;
    int trouve;

    if (L != NULL)
    {
        tmp = L;
        trouve = 0;
        while ((tmp != NULL)&&(trouve == 0))
        {
            if (tmp->Info == val)
                trouve = 1;
            else
                tmp = tmp->Suivant;
        }
        if (trouve == 1)
            printf(" %d EST DANS LA LISTE ", val);
        else
            printf(" %d N'EST PAS DANS LA LISTE ", val);
    }
    else
    {
        printf("LA LISTE EST VIDE\n");
    }
}
```

```

int main()
{
    Tete = NULL ;
    printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
    scanf("%s", Valeur);
    while (atoi(Valeur) != -1)
    {
        (p) = malloc(sizeof(Element));
        p->Info = atoi(Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
        scanf("%s", Valeur);
    }
    printf("\n");
    printf(" RECHERCHE DE L'ELEMENT 5 : \n");
    printf("\n");
    RechercherValeurListe(Tete, 5);
    return 0;
} OK

```

## Traduction en C (Liste de chaîne de caractères)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
Typedef struct Element
{
    char Info[10];
    struct Element* Suivant;
}Element;
```

```
TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];
```

```
/*
*****
* Recherche la valeur val dans la Liste L *
*****
*/
void RechercherValeurListe(Liste L, char val[10])
{
    Liste tmp;
    int trouve;

    if (L != NULL)
    {
        tmp = L;
        trouve = 0;
        while ((tmp != NULL) && (trouve == 0))
        {
            if (strcmp(tmp->Info, val) == 0)
                trouve = 1;
            else
                tmp = tmp->Suivant;
        }
        if (trouve == 1)
            printf(" %s EST DANS LA LISTE ", val);
        else
            printf(" %s N'EST PAS DANS LA LISTE ", val);
    }
    else
    {
        printf("LA LISTE EST VIDE\n");
    }
}
```

```

int main()
{
    Tete = NULL ;
    printf("SAISIR UNE CHAINE ('x' POUR FINIR) : ");
    scanf("%s", Valeur);
    while (Valeur[0] != 'x')
    {
        (p) = malloc(sizeof(Element));
        strcpy(p->Info, Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UNE CHAINE ('x' POUR FINIR) : ");
        scanf("%s", Valeur);
    }
    printf("\n");
    printf(" RECHERCHE DE L'ELEMENT konan : \n");
    printf("\n");
    RechercherValeurListe(Tete, "konan");
    return 0;
} OK

```

### 2.3.5. Supprimer le premier élément d'une Liste chaînée

Il y a deux actions, dans cet ordre, à réaliser :

- faire pointer la tête de **Liste** sur le deuxième élément de la **Liste**,
- libérer l'espace mémoire occupé par l'élément supprimé.

Il est nécessaire de déclarer un pointeur local qui va pointer sur l'élément à supprimer, et permettre de libérer l'espace qu'il occupait.



**PROCEDURE** SupprimerPremierElement((E/S)Tete : **Liste**)

/\* Supprime le premier élément de la **Liste** dont le pointeur de tête est passé en paramètre \*/

**VARIABLES**

P : **Liste** /\* pointeur sur l'élément à supprimer \*/

**DEBUT**

**SI** Tete <> Nil **ALORS**/\* la **Liste** n'est pas vide on peut donc supprimer le premier élément \*/

P ← Tete /\* P pointe sur le 1er élément de la **Liste**\*/

Tete ← P^.Suivant /\* la tête de **Liste** doit pointer sur le deuxième 'élément \*/

**DESALLouer**(P) /\* libération de l'espace mémoire qu'occupait le premier élément \*/

**SINON**

**ECRIRE**("La **Liste** est vide")

**FINSI**

**FIN**

## Traduction en C(Liste d'ENTIERs)

```
#include <stdio.h>
#include <stdlib.h>
```

```
Typedef struct Element
{
    int Info;
    struct Element* Suivant;
}Element;
```

```
TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];
```

```
/******
* Supprime l'élément en tête de Liste*
******/
Liste SupprimerPremierElement(Liste L)
{
    Liste Suivant = L;
    if (L != NULL)
    {
        // POUR etre sûr que L->Suivant existe
        Suivant = L->Suivant;
        free(L); //libération de l'espace alloué
    }
    return Suivant;
}
```

```
/******
* Affiche le contenu de la Liste L *
******/
void AfficherListe(Liste L)
{
    Liste tmp = L;
    while (tmp != NULL)
    {
        printf("->%d",tmp->Info);
        tmp = tmp->Suivant;
    }
    printf("->NULL\n");
}
```

```

int main()
{
    Tete = NULL ;
    printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
    scanf("%s", Valeur);
    while (atoi(Valeur) != -1)
    {
        (p) = malloc(sizeof(Element));
        p->Info = atoi(Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
        scanf("%s", Valeur);
    }
    printf("\n");
    printf(" AFFICHAGE DE LA LISTE : \n");
    printf("\n");
    AfficherListe(Tete);
    printf("\n");
    printf(" SUPPRESSION DU PREMIER ELEMENT : \n");
    Tete = SupprimerPremierElement(Tete);
    printf("\n");
    printf(" AFFICHAGE DE LA LISTE RESTANTE : \n");
    printf("\n");
    AfficherListe(Tete);
    return 0;
} OK

```

## Traduction en C (Liste de chaîne de caractères)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
```

```
Typedef struct Element
{
    char Info[10];
    struct Element* Suivant;
}Element;
```

```
TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];
```

```
/******
* Supprime l'élément en tête de Liste*
******/
Liste SupprimerPremierElement(Liste L)
{
    Liste Suivant = L;
    if (L != NULL)
    {
        // POUR etre sûr que L->Suivant existe
        Suivant = L->Suivant;
        free(L); //libération de l'espace alloué
    }
    return Suivant;
}
```

```
/******
* Affiche le contenu de la Liste L *
******/
void AfficherListe(Liste L)
{
    Liste tmp = L;
    while (tmp != NULL)
    {
        printf("->%s",tmp->Info);
        tmp = tmp->Suivant;
    }
    printf("->NULL\n");
}
```



```

int main()
{
    Tete = NULL ;
    printf("SAISIR UNE CHAINE ('x' POUR FINIR) : ");
    scanf("%s", Valeur);
    while (Valeur[0] != 'x')
    {
        (p) = malloc(sizeof(Element));
        strcpy(p->Info, Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UNE CHAINE ('x' POUR FINIR) : ");
        scanf("%s", Valeur);
    }
    printf("\n");
    printf(" AFFICHAGE DE LA LISTE : \n");
    printf("\n");
    AfficherListe(Tete);
    printf("\n");
    printf(" SUPPRESSION DU PREMIER ELEMENT : \n");
    Tete = SupprimerPremierElement(Tete);
    printf("\n");
    printf(" AFFICHAGE DE LA LISTE RESTANTE : \n");
    printf("\n");
    AfficherListe(Tete);
    return 0;
} OK

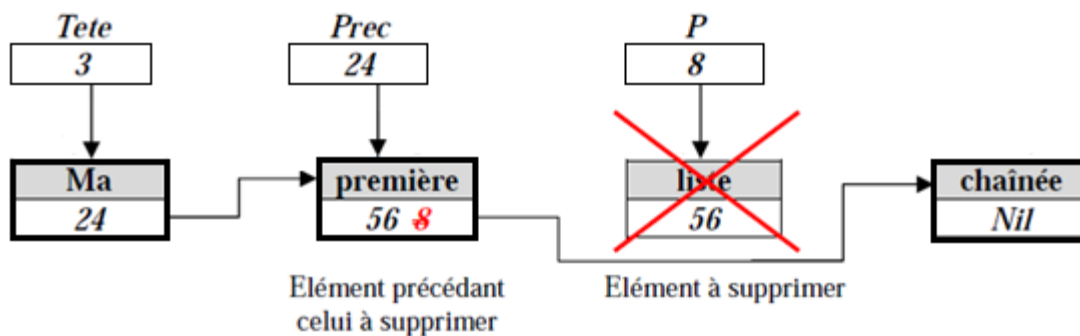
```

### 2.3.6. Supprimer d'une Liste chaînée un élément portant une valeur donnée

Il faut:

- traiter à part la suppression du premier élément car il faut modifier le pointeur de tête,
- trouver l'adresse P de l'élément à supprimer,
- sauvegarder l'adresse **Prec** de l'élément précédant l'élément pointé par **P** pour connaître l'adresse de l'élément précédant l'élément à supprimer, puis faire pointer l'élément précédent sur l'élément suivant l'élément à supprimer,
- Libérer l'espace mémoire occupé par l'élément supprimé.

L'exemple considère que l'on souhaite supprimer l'élément contenant la valeur "**Liste**" de la Liste ci-dessus.



**PROCEDURE** SupprimerElement((E/S) Tete : **Liste**, (E )Val : **chaîne**)

/\* Supprime l'élément dont la valeur est passée en paramètre \*/

**VARIABLES**

P : **Liste** /\* pointeur sur l'élément à supprimer \*/

Prec : **Liste** /\* pointeur sur l'élément précédant l'élément à supprimer \*/

Trouvé : Booleen /\* indique si l'élément à supprimer a été trouvé \*/

**DEBUT**

**SI** Tete <> Nil **ALORS**/\* la **Liste** n'est pas vide on peut donc y  
chercher une valeur à supprimer \*/

**SI** Tete^.info = Val **ALORS**/\* l'élément à supprimer est le premier \*/

P ← Tete

Tete ← Tete^.Suivant

**DESALLOUER**(P)

**SINON**/\* l'élément à supprimer n'est pas le premier \*/

Trouve ← Faux

Prec ← Tete /\* pointeur précédent \*/

P ← Tete^.Suivant /\* pointeur courant \*/

**TANTQUE** P <> Nil **ET** Non Trouve **FAIRE**

**SI** P^.Info = Val **ALORS**/\* L'élément recherché est l'élément courant \*/

Trouve ← Vrai

**SINON**/\* L'élément courant n'est pas l'élément cherché \*/

Prec ← P /\* on garde la position du précédent \*/

P ← P^.Suivant /\* on passe à l'élément suivant dans la **Liste**\*/

**FINSI**

**FINTANTQUE**

**SI** Trouve **ALORS**

Prec ← P^.Suivant /\* on "saute" l'élément à supprimer \*/

**DESALLOUER**(P)

Tete^.Suivant ← Prec

**SINON**

**ECRIRE**("La valeur ", Val, " n'est pas dans la **Liste**")

**FINSI**

**FINSI**

**SINON**

**ECRIRE**("La **Liste** est vide")

**FINSI**

**FIN**

## Traduction en C (Liste d'ENTIERs)

```
#include <stdio.h>
#include <stdlib.h>

Typedef struct Element
{
    int Info;
    struct Element* Suivant;
}Element;

TypedefElement*Liste;
Liste Tete, p ;
char Valeur[10];

/*****
* Suppression de la valeur val dans la Liste L *
*****/
void SupprimerElement(Liste L,int val)
{
    Liste tmp, prec;
    int trouve;

    if (L != NULL)
    {
        if (L->Info == val)
        {
            tmp = L ;
            L = L->Suivant;
            free(L); //libération de l'espace alloué
        }
        else
        {
            trouve = 0;
            prec = L ; // pointeur précédent
            tmp = L->Suivant; // pointeur courant
            while ((tmp != NULL) && (trouve == 0))
            {
                if (tmp->Info == val)
                {
                    trouve = 1;
                }
                else
                {
                    // on garde le precedent
                    prec = tmp ;
                    // on passe au suivant
                    tmp = tmp->Suivant;
                }
            }
            if (trouve == 1)
            {
                // on saute l'élément à supprimer
                prec = tmp->Suivant;
                //libération de l'espace alloué
            }
        }
    }
}
```

```

        free(tmp);
        L->Suivant = prec ;
        printf("L'ELEM. %i EST SUPPRIME ", val);
    }
    else
    {
        printf(" %d PAS DS LA LISTE ", val);
    }
}

else
{
    printf("LA LISTE EST VIDE\n");
}

}

int main()
{
    Tete = NULL ;
    printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
    scanf("%i", Valeur);
    while (atoi(Valeur) != -1)
    {
        (p) = malloc(sizeof(Element));
        p->Info = atoi(Valeur);
        p->Suivant = Tete;
        Tete = p ;
        printf("SAISIR UN ENTIER(-1 POUR FINIR) : ");
        scanf("%i", Valeur);
    }
    printf("\n");
    printf(" SUPPRESSION DE L'ELEMENT 5 : \n");
    printf("\n");
    SupprimerElement(Tete, 5);
    return 0;
} OK

```

## 2.4. Listes doublement chaînées

Il existe aussi des Listes chaînées, dites bidirectionnelles, qui peuvent être parcourues dans les deux sens, du 1er élément au dernier et inversement.

Une Liste chaînée bidirectionnelle est composée :

- d'un ensemble de données,
- de l'ensemble des adresses des éléments de la Liste,
- d'un ensemble de pointeurs *Suivant* associés chacun à un élément et qui contient l'adresse de l'élément suivant dans la Liste,
- d'un ensemble de pointeurs *Precedent* associés chacun à un élément et qui contient l'adresse de l'élément précédent dans la Liste,
- du pointeur sur le premier élément *Tete*, et du pointeur sur le dernier élément, *Queue*,

**TYPE ListeDC = ^Element**

**TYPE Element = STRUCTURE**

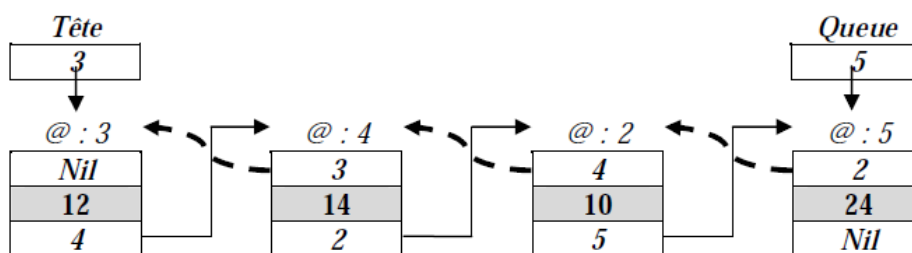
Precedent : ListeDC

Info : variant

Suivant : ListeDC

**FINSTRUCTURE**

Le pointeur *Precedent* du premier élément ainsi que le pointeur *Suivant* du dernier élément contiennent la valeur *Nil*.



A l'initialisation d'une Liste doublement chaînée les pointeurs *Tete* et *Queue* contiennent la valeur *Nil*.

### Afficher les éléments d'une Liste doublement chaînée

Il est possible de parcourir la Liste doublement chaînée du premier élément vers le dernier. Le pointeur de parcours, P, est initialisé avec l'adresse contenue dans *Tete*. Il prend les valeurs successives des pointeurs *Suivant* de chaque élément de la Liste. Le parcours s'arrête lorsque le pointeur de parcours a la valeur *Nil*. Cet algorithme est analogue à celui du parcours d'une Liste simplement chaînée.

**PROCEDURE** AfficherListeAvant ((E) Tete : ListeDC)

**VARIABLES**

P : ListeDC

**DEBUT**

P ← Tete

**TANTQUE** P <> NIL **FAIRE**

**ECRIRE**(P^.Info)

    P ← P^.Suivant

**FINTANTQUE**

**FIN**

Il est possible de parcourir la Liste doublement chaînée du dernier élément vers le premier. Le pointeur de parcours, P, est initialisé avec l'adresse contenue dans *Queue*. Il prend les valeurs successives des pointeurs *Precedent* de chaque élément de la Liste. Le parcours s'arrête lorsque le pointeur de parcours a la valeur *Nil*.

**PROCEDURE** AfficherListeArriere ((E) Queue : ListeDC)

**VARIABLES**

P : ListeDC

**DEBUT**

P ← Queue

**TANTQUE** P <> NIL **FAIRE**

**ECRIRE**(P^.Info)

    P ← P^.Precedent

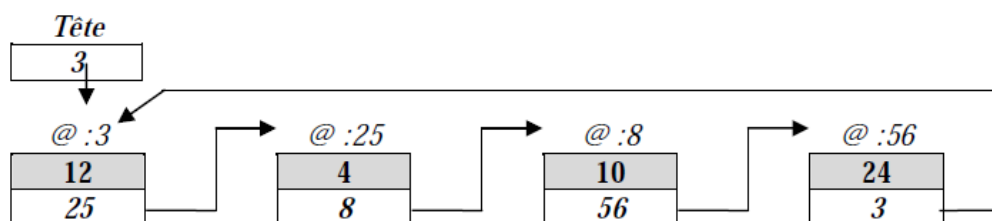
**FINTANTQUE**

**FIN**

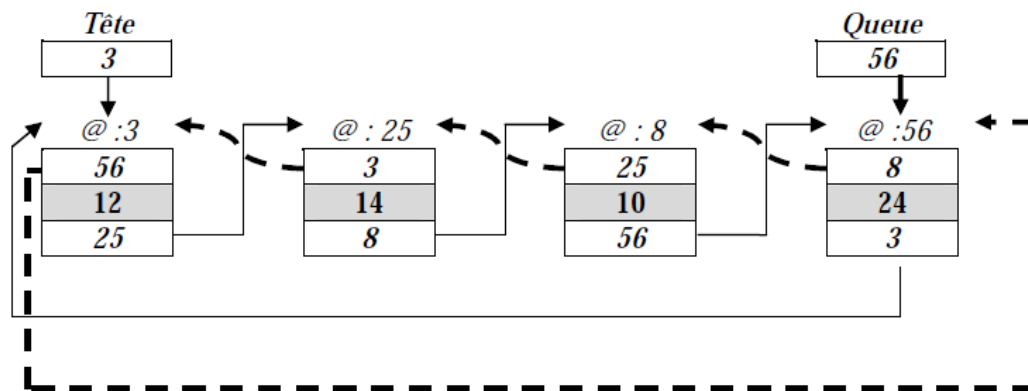
### 2.5. Listes chaînées circulaires

Une Liste chaînée peut être circulaire, c'est à dire que le pointeur du dernier élément contient l'adresse du premier.

Ci-dessous l'exemple d'une Liste simplement chaînée circulaire : le dernier élément pointe sur le premier.



Puis l'exemple d'une Liste doublement chaînée circulaire. : Le dernier élément pointe sur le premier, et le premier élément pointe sur le dernier.





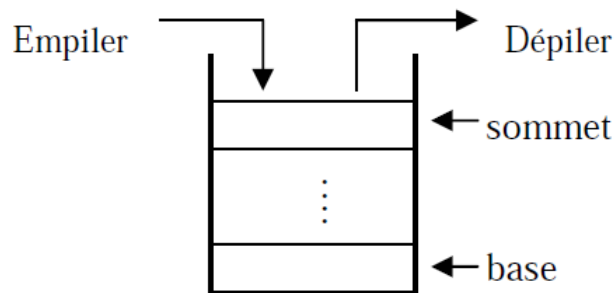
# Chapitre 3 : Piles et files

## 1. Piles

Une pile est une **Liste** chaînée d'informations dans laquelle :

- Un élément ne peut être ajouté qu'au sommet de la pile,
- Un élément ne peut être retiré que du sommet de la pile.

Il s'agit donc d'une structure de **TYPE** LIFO (Last In First Out). On ne travaille que sur le sommet de la pile. Les piles sont comparables à des piles d'assiettes.



On associe à une pile les termes de :

- PUSH pour empiler c'est-à-dire ajouter un élément,
- POP pour dépiler c'est-à-dire supprimer un élément.

Les piles servent à revenir à l'état précédent et sont utilisées pour :

- Implanter les appels de procédures (pour revenir à l'état d'avant l'appel),
- Annuler une commande,
- Evaluer des expressions arithmétiques,
- etc.

### 1.1. Opérations autorisées

Les opérations autorisées avec une pile sont :

- Empiler, toujours au sommet, et jusqu'à la limite de la mémoire,
- Dépiler, toujours au sommet, si la pile n'est pas vide,
- Vérifier si la pile est vide ou non.

On peut implémenter une pile dans un tableau (pile statique) ou dans une **Liste** chaînée (pile dynamique). C'est l'implémentation en **Liste** chaînée qui est présentée ici. Le sommet de la pile est le premier élément et le pointeur de tête pointe sur ce sommet. Il faut commencer par définir un **TYPE** de variable pour chaque élément de la pile. La déclaration est identique à celle d'une **Liste** chaînée, par exemple pour une pile de chaînes de caractères :

**TYPE** Pile = ^**Element**

**TYPE** **Element** = **STRUCTURE**

Info : chaîne de caractères

Suivant : Pile

**FINSTRUCTURE**

### 1.1.1. Empiler

Empiler un élément revient à faire une insertion en tête dans la **Liste** chaînée.

**PROCEDURE** Empiler (*Entrée/Sortie* Tête : Pile, *Entrée* Valeur : chaîne de caractères)

*/\* Ajout d'un élément dans une pile passée en paramètre \*/*

**Variable locale**

P : Pile */\* pointeur auxiliaire \*/*

**DEBUT**

*/\* Réserve un espace mémoire pour le nouvel élément \*/*

**1 ALLOUER(P)**

*/\*stocke dans l'Info de l'élément pointé par P la valeur passée en paramètre \*/*

**2** P^.Info ← Valeur

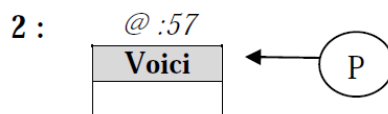
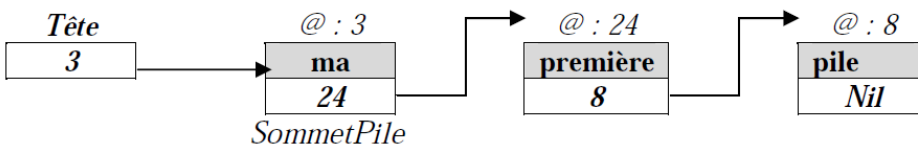
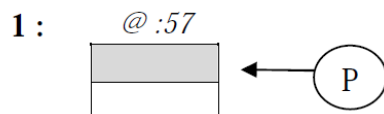
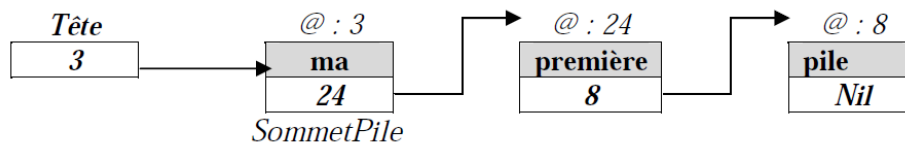
*/\*stocke dans l'adresse du Suivant l'adresse de Tête \*/*

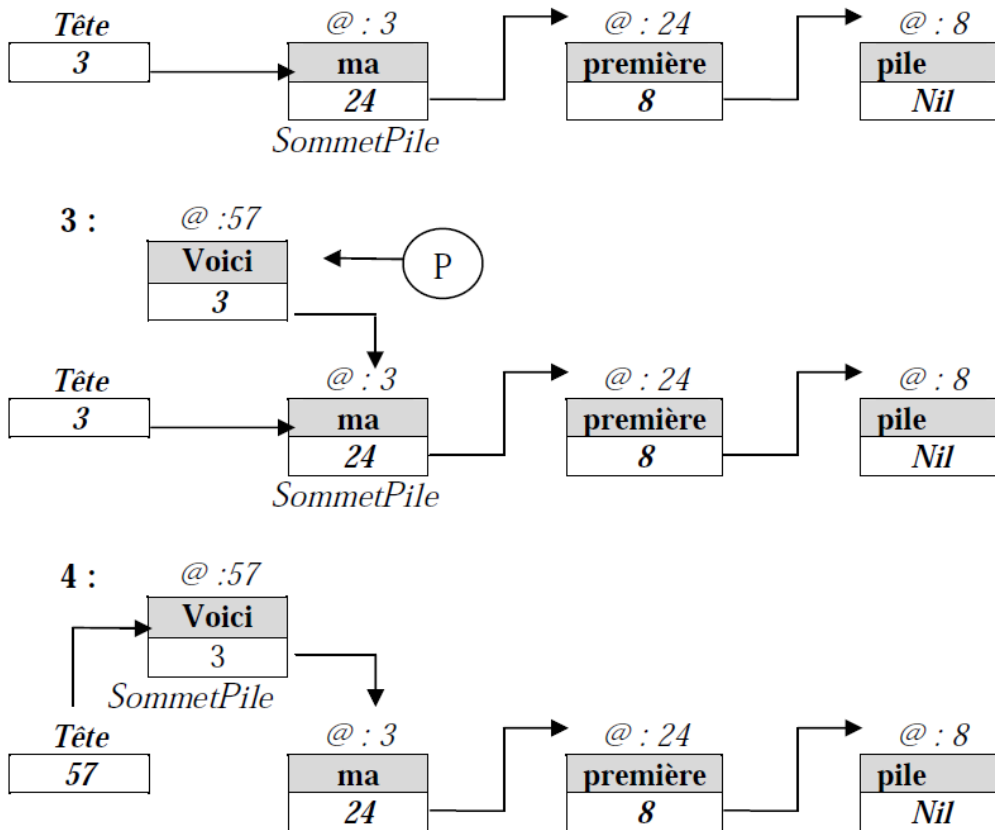
**3** P^.Suivant ← Tête

*/\* Tête pointe maintenant sur le nouvel élément \*/*

**4** Tête ← P

**FIN**





En faisant référence à la procédure *InsererEnTete* dans les exercices à propos des **Listes** chaînées simples, nous pouvons écrire :

**PROCEDURE** Empiler ((E/S) Tête : Pile; Entrée **Element** : chaîne de caractères)

/\* Ajout de l'élément au sommet de la pile P passée en paramètre \*/

**DEBUT**

*InsererEnTete* (Tête, **Element**)

**FIN**

Puisqu'il s'agit d'une gestion dynamique de la mémoire, il n'est pas nécessaire de vérifier si la pile est pleine.

### 1.1.2. Dépiler

Dépiler revient à faire une suppression en tête.

**PROCEDURE** Dépiler (Entrée/Sortie Tête : Pile)

/\* Suppression de l'élément au sommet de la pile passée en paramètre \*/

**VARIABLES locale**

P : Pile /\* Pointeur nécessaire pour libérer la place occupée par l'élément dépiler \*/

**DEBUT**

/\* Vérifier si la pile est vide \*/

**SI** Tête <> NIL **ALORS** /\* la pile n'est pas vide donc on peut dépiler \*/

P ← Tête /\* on garde l'adresse du sommet pour désallouer \*/

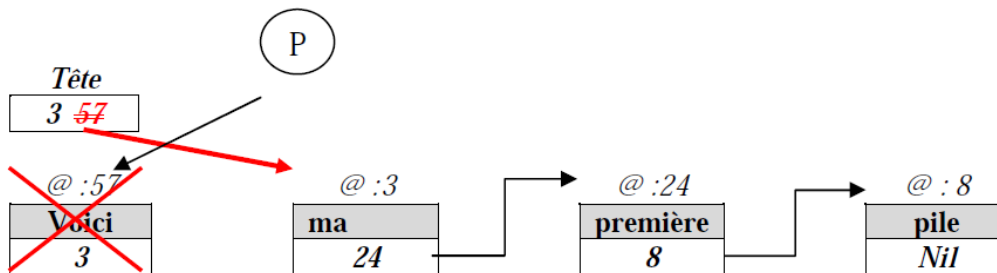
Tête ← Tête^.Suivant /\* P va pointer sur le 2ème élément de la pile qui devient le sommet \*/

**DESALLOUER**(P)

**FINSI**

**FIN**

Si la pile ne contient qu'un élément, *Tête* prend la valeur *Nil* et la nouvelle pile est vide.



### 1.1.3. Procédures et fonctions de base

**PROCEDURE** InitPile ((S) Tête : Pile)

/\* Initialise la création d'une pile \*/

**DEBUT**

Tête ← Nil

**FIN**

**FONCTION** PileVide (Tête : Pile) : booléen

/\* indique si la pile est vide ou non \*/

**DEBUT**

Retourner (Tête = Nil)

**FIN**

**FONCTION** SommetPile (Tête : Pile) : variant

/\* renvoie la valeur du sommet si la pile n'est pas vide \*/

**DEBUT**

SommetPile ← Tête^.Info

**FIN**

## 2. Files

Une file, ou file d'attente, est une **Liste** chaînée d'informations dans laquelle :

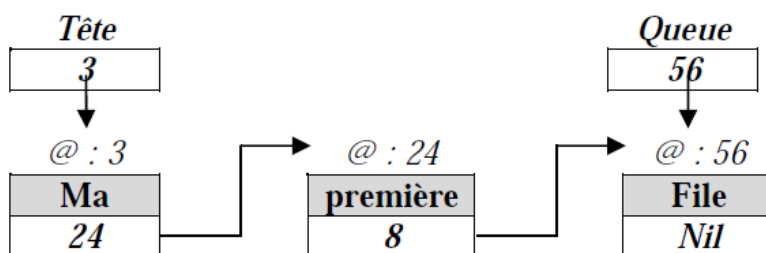
- Un élément ne peut être ajouté qu'à la queue de la file,
- Un élément ne peut être retiré qu'à la tête de la file.

Il s'agit donc d'une structure de **TYPE** FIFO (First In First Out). Les données sont retirées dans l'ordre où elles ont été ajoutées. Une file est comparable à une queue de clients à la caisse d'un magasin.

Les files servent à traiter les données dans l'ordre où on les a reçues et permettent de :

- Gérer des processus en attente d'une ressource système (par exemple la **Liste** des travaux à éditer sur une imprimante)
- Construire des systèmes de réservation
- etc.

Pour ne pas avoir à parcourir toute la **Liste** au moment d'ajouter un élément en queue, on maintient un pointeur de queue. Attention une file peut très bien être simplement chaînée même s'il y a un pointeur de queue.



### 2.1. Opérations autorisées

Les opérations autorisées avec une file sont :

- Enfiler toujours à la queue et jusqu'à la limite de la mémoire,
- Défiler toujours à la tête si la file n'est pas vide,
- Vérifier si la file est vide ou non.

On peut implémenter une file dans un tableau (file statique) ou dans une **Liste** chaînée (file dynamique). C'est l'implémentation en **Liste** chaînée qui est présentée ici. Le pointeur de tête pointe sur le premier élément de la file, et le pointeur de queue sur le dernier. Il faut commencer par définir un **TYPE** de variable pour chaque élément de la file. La déclaration est identique à celle d'une **Liste** chaînée, par exemple pour une file de chaînes de caractères :

**TYPE** File = ^Element

**TYPE** Element = STRUCTURE

Info : chaîne de caractères

Suivant : File

**FINSTRUCTURE**

### 2.1.1. Enfiler

Enfiler un élément consiste à l'ajouter en queue de **Liste**. Il faut envisager le cas particulier où la file était vide. En effet, dans ce cas, le pointeur de tête doit être modifié.

**PROCEDURE** Enfiler ((E/S) : Tête, Queue : File, Entrée Valeur : chaîne de caractères)

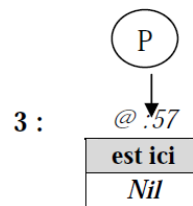
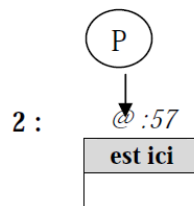
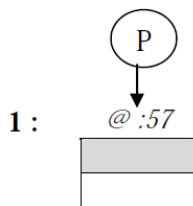
/\* Ajout d'un élément dans une file \*/

#### VARIABLES

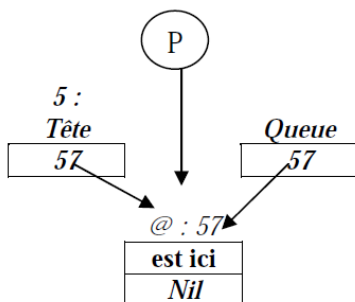
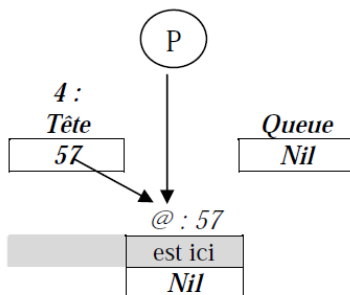
P : File /\* Pointeur nécessaire pour allouer la place au nouvel élément \*/

#### DEBUT

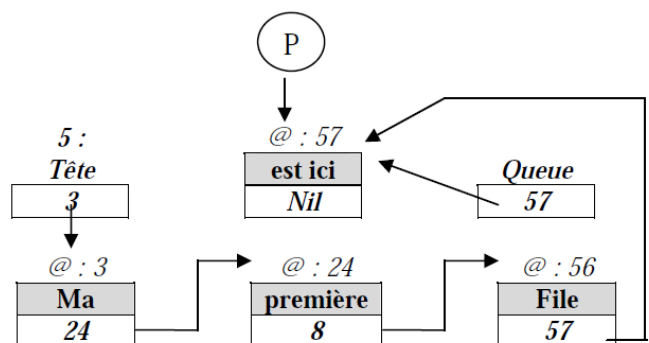
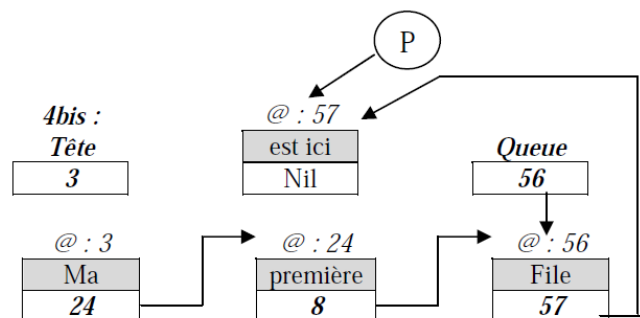
- 1 **ALLOUER**(P) /\* Réserve un espace mémoire pour le nouvel élément \*/
  - 2 P^.Info ← Valeur /\* stocke la valeur dans l'Info de l'élément pointé par P \*/
  - 3 P^.Suivant ← Nil /\* stocke Nil (ce sera le dernier de la file) dans Suivant \*/  
**SI** Tête = Nil **ALORS** /\* file vide \*/
    - 4 Tête ← P /\* Tête pointe maintenant sur l'élément unique \*/**SINON** /\* il y a au moins un élément \*/
    - 4bis Queue^.Suivant ← P /\* le nouvel élément est ajouté au dernier \*/
  - FINSI**
  - 5 Queue ← P /\* Queue pointe maintenant sur l'élément ajouté \*/
- FIN**



Si Tête = Nil



Si Tête ≠ Nil



En faisant référence à la procédure *InsererEnTete* dans les exercices sur des **Listes** chaînées simples, nous pouvons écrire :

**PROCEDURE** Enfiler (*Entrée/Sortie* : Tête, Queue : File, *Entrée* Valeur : chaîne de caractères)  
/\* Ajout d'un élément dans une file \*/

**DEBUT**

**SI** Tete = Nil **ALORS**/\* Tete = Nil et Queue = Nil \*/

*InsererEnTete* (Tete, Valeur)

        Queue ← Tete

**SINON**

*InsererEnTete* (Queue^.Suivant, Valeur)

        Queue ← Queue^.Suivant

**FINSI**

**FIN**

### 2.1.2. Défiler

Défiler est équivalent à dépiler et consiste à supprimer l'élément de tête si la file n'est pas vide. Si la file a un seul élément, il faut mettre à jour le pointeur de queue car on vide la file. Il faut conserver l'adresse de l'élément qu'on supprime pour libérer sa place.

**PROCEDURE** Défiler ((E/S) Tête, Queue : File, (S) Valeur : chaîne)

/\* Suppression de l'élément de tête de la file passée en paramètre \*/

**VARIABLE locale**

    P : File /\* Pointeur nécessaire pour libérer la place de l'élément supprimé \*/

**DEBUT**

**SI** Tête <> NIL **ALORS**/\* la **Liste** n'est pas vide donc on peut défiler \*/

        Valeur ← Tête^.Info /\* on récupère l'élément de tête \*/

        P ← Tête /\* on garde l'adresse du sommet pour désallouer \*/

        Tête ← Tête^.Suivant /\* P va pointer sur le 2ème élément de la pile  
                                    qui devient le sommet \*/

**DESALLOUER**(P)

**SI** Tête = Nil **ALORS**

        Queue ← Nil /\* la file a été vidée \*/

**FINSI**

**FINSI**

**FIN**

En faisant référence à la procédure *SupprimerEnTete* dans les exercices à propos des **Listes** chaînées simples, nous pouvons écrire :

**PROCEDURE** Défiler ((E/S) Tête, Queue : File, (S) Val : chaîne de caractères)

/\* Suppression d'un élément dans une file \*/

**DEBUT**

**SI** Tete <> Nil **ALORS**

        Val ← Tete^.Info

*SupprimerEnTete* (Tete)

**SI** tête = Nil **ALORS**

        Queue ← Nil

**FINSI**

**FINSI**

**FIN**



## UN EXEMPLE COMPLET QUI MARCHE

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
/* DÉFINITION de la structure cellule*/
struct cellule
{
    int info;
    struct cellule *suivant;
};
//TYPE Liste synonyme du TYPE pointeur vers une struct cellule
typedef struct cellule *Liste;

/*****
* Insère un élément en tête de Liste*
*****/
Liste insere(int Element,Liste Q)
{
    Liste L;
    L = (Liste)malloc(sizeof(struct cellule)); // allocation mémoire
    L->info = Element;
    L->suivant = Q;
    return L;
}

/*****
* Insère un élément en queue de Liste*
*****/
Liste insereInTail(int Element,Liste Q)
{
    Liste L, tmp = Q;
    L = (Liste)malloc(sizeof(struct cellule)); // allocation mémoire
    L->info = Element;
    L->suivant = NULL;
    if (Q == NULL)
        return L;
    // maintenant tmp = Q est forcément non vide => tmp->suivant existe
    while (tmp->suivant != NULL) tmp = tmp->suivant;
    // déplacement jusqu'au dernier elt de la Liste
    tmp->suivant = L;
    return Q;
}

/*****
* Supprime l'élément en tête de Liste*
*****/
Liste supprime_tete(Liste L)
{
    Liste suivant = L;
    if (L != NULL)
    {
        // POUR être sûr que L->suivant existe
        suivant = L->suivant;
        free(L); //libération de l'espace alloué POUR une cellule
    }
    return suivant;
}

/*****
* Supprime toute la Liste L *
*****/
Liste supprime(Liste L)
{
    while (L != NULL)
        L = supprime_tete(L); //suppression de la tête de Liste
    return L;
}
```

```

/*****
* Affiche le contenu de la Liste L *
*****/
void print_Liste(Liste L)
{
    Liste tmp = L;
    while (tmp != NULL)
    { //on aurait pu écrire 'while (tmp)'
        printf("%d \t",tmp->info);
        tmp = tmp->suivant;
    }
    printf("NULL\n");
}
/*****/
int main()
{
    Liste L;
    L = insere(14,insere(2,insere(3,insere(10,NULL))));
    L = insereInTail(15,(insereInTail(14,L)));
    printf("Impression de la Liste:\nL = \t");
    print_Liste(L);
    printf("Suppression de l'Element en tete:\nL = \t");
    L = supprime_tete(L);
    print_Liste(L);
    printf("Suppression de la Liste:\nL = \t");
    L = supprime(L);
    print_Liste(L);
    return 0;
}

```