

LEÇON 5 : Les Listes chaînées

UVCI

Table des matières



I - 1- Généralité sur les listes chaînées	3
II - Application 1 :	6
III - Exercice	7
IV - 2- Traitement sur les listes chaînées	8
V - Application 2 :	15

1- Généralité sur les listes chaînées

I

🔑 Définition : 1.1- Définition

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*.

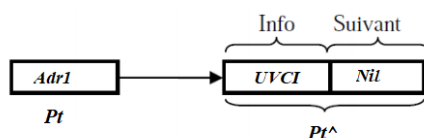
Un élément d'une Liste est l'ensemble (ou structure) formé :

- d'une *donnée ou information*,
- d'un *pointeur 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 (pointeur).

Exemple :



Explication : La variable pointeur *Pt* pointe sur l'espace mémoire *Pt^* d'adresse *Adr1*. Cette cellule mémoire contient la valeur "UVCI" dans le champ *Info* et la valeur spéciale *Nil* dans le champ *Suivant*.

Ce champ *Suivant* 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. *Pt^* est l'objet dont l'adresse est rangée dans *Pt*.

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

- *Allouer(Pt)* : réserve un espace mémoire *Pt^* et donne pour valeur à *Pt* l'adresse de cet espace mémoire. On alloue un espace mémoire pour un élément sur lequel pointe *Pt*.
- *Désallouer(Pt)* : libère l'espace mémoire qui était occupé par l'élément à supprimer *Pt^* sur lequel pointe *Pt*.

Pour définir les variables utilisées ci-dessus, il faut :

//définir le type des éléments de liste :

Type Cellule= Structure

Info : Chaîne

Suivant : Liste

fin Structure

//définir le type du pointeur :

Type Liste = ^Cellule

//déclarer une variable pointeur à partir du type pointeur transcrit dans Liste :

Var

Pt : Liste

//allouer une cellule mémoire qui réserve un espace en mémoire et donne à Pt la valeur de l'adresse de l'espace mémoire Pt^:

Allouer(P)

//affecter des valeur à l'espace mémoire Pt^:

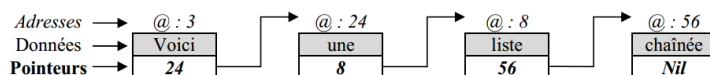
Pt^.Info ← "UVCI"

Pt^.Suivant ← Nil

Quand Pt = Nil alors Pt ne pointe sur rien.

1.2- Représentation d'une liste chaînée

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 etc.

1.3- Les Types de listes chaînées

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 de premier élément pointe également sur le dernier.

Ces différents types peuvent être mixés selon les besoins.

NB :

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.

Application 1 :



Exercice

Une liste chaînée est :

- ☐ est une structure linéaire ayant une dimension fixée à sa création.
- ☐ est une structure linéaire qui conserve que les adresses.
- ☐ est une structure linéaire n'ayant pas de dimension fixée à sa création.

Exercice

Pour accéder à une liste il faut :

- ☐ passer par la fin de la liste
- ☐ passer par la tête de la liste
- ☐ passer par la fin de la liste



2- Traitement sur les listes chaînées

IV

2.1- Les listes chaînées simple

Les traitements des listes sont les suivants :

2.1.1- Créer une liste chaînée

Syntaxe :

Type Nom_structure = Structure

champ1 : type1

Suivant : Nom_Liste

Finstructure

Type Nom_Liste = ^Nom_structure

Exemple :

Déclarer une liste ayant un champ valeur.

Type message = Structure

valeur : chaîne

svt : Liste

Finstructure

Type Liste = ^message

Application :

Créer une liste message composée de 2 éléments de type chaîne de caractères.

Algorithme liste1

///***** Déclaration de la liste *****/

Type message = Structure

valeur : chaîne

svt : Liste

Finstructure

Type Liste = ^message

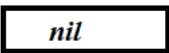
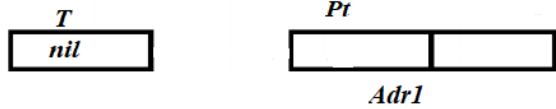
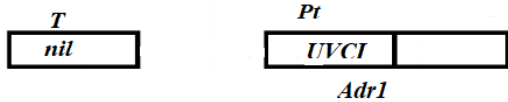
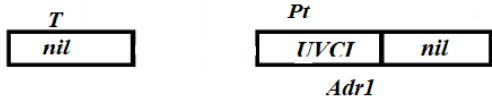
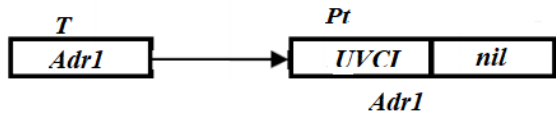
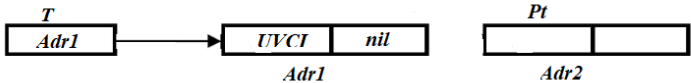

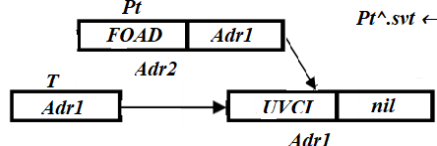
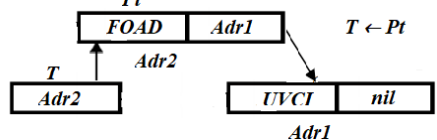

```

var
//***** Déclaration des variable pointeurs T pour la tête de la liste et Pt pour le contenu de la
liste*****
T, Pt : Liste
i : entier
Début
T ← nil //*** La tête de la liste est vide au départ
Pour i ← 1 à 2 faire //*****Utilisation de la boucle pour créer les deux éléments de la liste
allouer(Pt) //***** réserve de l'espace en mémoire à chaque fois que i aura une nouvelle valeur
afficher "Donner la valeur n° ",i," de la liste " //***** envoie un message pour inviter l'utilisateur à entrer une
valeur
saisir Pt^.valeur //***** stocke dans valeur de l'élément pointé par Pt la valeur saisie
Pt^.svt ← T //***** stocke dans svt de l'élément pointé par Pt la valeur de T
T ← Pt //***** T récupère l'adresse de Pt
finpour
fin

```

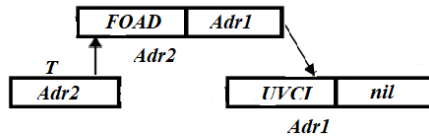
Explication :

Ce tableau ci-dessous décrit la manière que les listes chaînées sont créer pour chaque itération de i en exécutant les instructions

i	Description de chaque instruction
Aucune valeur	$T \leftarrow nil$ 
i=1	$allouer(Pt)$ 
i=1	$saisir Pt^{\wedge}.valeur$ 
i=1	$Pt^{\wedge}.svt \leftarrow T$ 
i=1	$T \leftarrow Pt$ 
i=2	$allouer(Pt)$ 
i=2	$saisir Pt^{\wedge}.valeur$ 
i=2	$Pt^{\wedge}.svt \leftarrow T$ 
i = 2	$T \leftarrow Pt$ 

2.1.2- 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.

Procédure AfficherListe ({E} P : Liste)

****** Afficher les éléments d'une liste chaînée passée en paramètre*

Début

P ← T ****** 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*

Tantque P <> nil faire ****** si la liste est vide Tete est à Nil*

Afficher P^.valeur ****** afficher la valeur contenue à l'adresse pointée par P*

P ← P^.svt ****** On passe à l'élément suivant*

fintq

fin

2.1.3- Rechercher une valeur donnée dans une liste chaînée ordonnée

L'algorithme est donné sous forme d'une procédure qui reçoit la tête de liste en paramètre et l'élément à rechercher.

Procédure RechercherValeurListe ({E} T : Liste, Val : chaîne)

****** Rechercher si une valeur donnée en paramètre est présente dans la liste passée en paramètre ******

Var

P : Liste ****** pointeur de parcours de la liste*

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

Début

Si T <> Nil Alors ****** la liste n'est pas vide on peut donc y chercher une valeur*

P ← T ****** initialise la liste de parcours par la liste passée en paramètre*

Trouve ← Faux ******Initialise Trouve à faux*

Tantque P <> Nil et Non(Trouve) faire ******Vérifie si nous ne sommes pas à la fin de la liste ou si nous avons retrouvé la valeur.*

si P^.valeur = Val alors ****** L'élément recherché est l'élément courant*

Trouve ← Vrai ****** Trouve reçoit la valeur vrai*

sinon ****** L'élément courant n'est pas l'élément recherché*

P ← P^.svt ****** on passe à l'élément suivant dans la liste*

finsi

fintq

si Trouve alors

Afficher " La valeur ", Val, " est dans la liste"

sinon

Afficher " La valeur ", Val, " n'est pas dans la liste"

finsi

sinon

Afficher "La liste est vide"

finsi

fin

2.1.4- Supprimer d'une liste chaînée un élément portant une valeur donnée

Pour cela 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 de l'élément précédant 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'algorithme est donné sous forme d'une procédure qui reçoit la tête de liste en paramètre et la valeur de l'élément à supprimer.

Procédure SupprimerElement ({E/S} T : Liste, Val : chaîne)

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

Var

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

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

Trouve : Booléen */****** indique si l'élément à supprimer a été trouvé*

Début

si T <> Nil alors */****** la liste n'est pas vide on peut donc y chercher une valeur à supprimer*

si T^.valeur = Val alors */****** l'élément à supprimer est le premier*

P ← T

T ← T^.svt

Desallouer(P)

sinon

Trouve ← Faux

Prec ← T */****** pointeur précédent*

P ← T^.svt */****** pointeur courant*

```

Tantque P <> Nil et Non(Trouve) faire
  si P^.valeur = 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^.svt //***** on passe à l'élément suivant dans la liste
  finsi
Fintq
si Trouve alors
  Prec^.svt ← P^.svt //***** on "saute" l'élément à supprimer
Desallouer(P)
sinon
  Afficher "La valeur ", Val, " n'est pas dans la liste"
fin
fin
fin
fin
fin
fin
fin

```

2.2- Listes doublement chaînées

Il existe aussi des liste 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 Précédent associés chacun à un élément et qui contient l'adresse de l'élément précédent dans la liste,
- du pointeur sur le premier élément Tête, et du pointeur sur le dernier élément, Queue

Syntaxe de création :

Type Message = Structure

pre : ListeDC

valeur : chaine

svt : ListeDC

FinStructure

Type ListeDC = ^Message

Le pointeur prec du premier élément ainsi que le pointeur svt du dernier élément 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 *T*.

Il prend les valeurs successives des pointeurs *svt* 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.

Procédure AfficherListeAvant ({E} T : ListeDC)

var

P : ListeDC

Début

P ← T

Tantque P <> nil faire

afficher P^.valeur

P ← P^.svt

fintq

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 *Q*. Il prend les valeurs successives des pointeurs *Prec* de chaque élément de la liste. Le parcours s'arrête lorsque le pointeur de parcours a la valeur *Nil*.

Procédure AfficherListeArriere ({E} Q: ListeDC)

Variables locales

P : ListeDC

Début

P ← Q

Tantque P <> nil faire

Afficher P^.valeur

P ← P^.Prec

fintq

fin

Application 2 :

V

Exercice

NB : Toutes les réponses doivent être en minuscule et sans espace.

Partie 1 :

Déclarer une liste resultat constituée de moyenne d'étudiants.

Solution :

```
_____ = _____
```

```
moyenne : _____
```

```
suiv : _____
```

```
_____
```

```
_____ ptliste = _____
```

Partie 2 :

Écrire une fonction qui renvoie le nombre d'éléments de la liste créer ci-dessus.

Solution :

```
_____ compte_element(_____ T : _____) : _____
```

```
var
```

```
nbreelt : _____
```

```
p : _____
```

Début

```
nbreelt ← _____
```

```
p ← _____
```

```
_____ faire
```

```
nbElt ← _____
```

```
_____ ← _____
```

```
fintq
```

```
_____
```

Fin