Structures chaînées – Listes chaînées

Introduction

Pour stocker et manipuler de grandes quantités de données, on a vu précédemment :
- des tableaux, qui permettent de ranger plusieurs données de même type,
- des classes, qui permettent de ranger plusieurs données de types différents.

Ces 2 structures présentent des caractéristiques qu'on peut voir comme des avantages et des inconvénients :

	Tableaux	Classes
Avantages	- une variable « tableau » représente plusieurs informations - on peut accéder directement à chaque information par l'indexation de la variable « tableau »	une information peut être complexe, c'est à dire qu'on peut avoir plusieurs types "mélangés", on parle d'**agrégation**.
Inconvénients	**le nombre d'informations c'est à dire la taille du tableau est fixé avant l'utilisation du tableau et** **- dans le cas de tableau statique cette taille n'est pas modifiable** **- dans la cas de tableau dynamique sa modification est coûteuse (en temps)**	une variable représente une et une seule information (complexe).

Le point mis en gras dans la grille précédente constitue un défaut très important dans le sens où il ne permet plus de garantir que les temps d'exécution d'un programme resteront proportionnel (quel que soit la fonction de proportionnalité) à la quantité de données.

On va maintenant s'intéresser à des structures qui permettent d'agglomérer un **nombre quelconque** d'informations, de **types quelconques**.

Présentation - Définition

Les listes chaînées sont des structures abstraites qui possèdent les caractéristiques :
- la structure ne **permet** l'**accès directement** qu'**à une seule information**, que l'on qualifiera de première ou début ou **tête de liste**,
- une information donne accès à sa **suivante**.

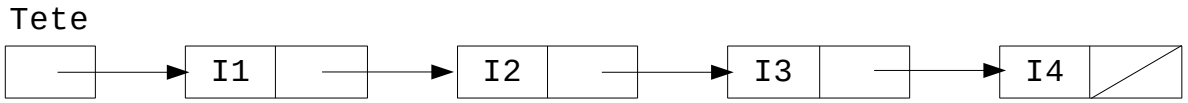
On dit aussi que la relation de successeur est explicite dans cette structure.

Les structures de listes chaînées sont souvent représentées graphiquement par des petits trains :

dont les wagons sont composés de 2 parties : la **première** porte **l'information** et la **deuxième** donne accès à l'information **suivante**. Le premier wagon est le seul accessible directement et constitue donc la tête de liste.
Pour marquer qu'il n'y a plus rien après le dernier wagon, on barre la dernière case.

Dans le vocabulaire consacré, le premier wagon est appelé **tête de liste** et les suivants sont appelés **maillons**. On peut noter que les suivants constituent une liste complète.

Le premier intérêt de cette structure est qu'il est possible de l'implémenter de manière à ce que **l'espace mémoire** occupé soit directement **proportionnel** à la **quantité de données** qu'elle contient et que sa croissance ne soit limité que par l'espace mémoire allouable : on crée une liste vide qui occupe peu d'espace et on ajoute des maillon au fur et à mesure des besoins.
L'inconvénient principal de cette structure concerne les possibilités d'accès aux données : on accède directement seulement à la première informations et pour accéder à une information quelconque, il faut d'abord accéder aux précédentes.



On accède directement à `I1`, pour accéder à `I3` il faut d'abord accéder à `I1`, puis à `I2` et enfin à `I3`, etc.

Attention, il faut toujours bien veiller à ne pas perdre le lien avec la 1re information, sinon c'est toutes les informations qui sont perdues !

Définition récursive

Une définition récursive des listes chaînées qui permettra de programmer récursivement les traitements sur les listes.

Une liste chaînées est soit :
- une liste vide
- une information suivie d'une liste

qui peut se formaliser en :
L est une liste si
- soit L = Φ
- soit L = (I, L')

avec Φ Liste vide, I une information, L' une liste.

Liste vide :
L = ()

1 élément :
L = (I1 , ())

2 éléments :
L = (I1 , (I2 , ())

...

L = (I1 , (I2 , (I3 , (I4 , (... , ()...))))

Cette définition récursive est à la base de traitement récursif sur des listes !

Implémentation en C++

Pour implémenter une liste chaînées il faut :
- représenter les maillons,
- un moyen facile de passer d'un maillon à l'autre,
- un moyen facile de créer et de supprimer des maillons,
- une structure qui conserve le début de la liste.

On va donc utiliser :
- une classe Maillon qui contiendra une information et un pointeur sur un maillon : le **suivant**,
- une classe Liste qui contiendra un pointeur sur le premier maillon : la **tête**.

```
class Maillon
{
  Type Info ;
  Maillon * suivant ;

  friend class Liste ;

} ;

class Liste
{
  Maillon * tete ;
  public :
  ...
} ;
```

Remarque :

Les données membres d'une classe sont par défaut "private", ce qui signifie qu'elles ne sont accessibles que par les fonctions membres de la classe. En particulier dans notre cas la donnée Info de Maillon n'est pas accessible par des fonctions de la classe Liste. Pour permettre cela on déclare dans la classe maillon que la classe Liste est une "amie" en ajoutant la ligne : `friend class Liste ;`

Cela permet aux fonctions de la classe Liste de manipuler (lire, modifier...) les données membres de la classe Maillon.

Représentation d'une liste en mémoire

En mémoire une liste est donc rangé dans des maillons qui seront créés au fur et à mesure des besoins :
au début la liste est vide et on ajoute les données, une par une.

Déclaration d'une liste :

```
Liste L;
```

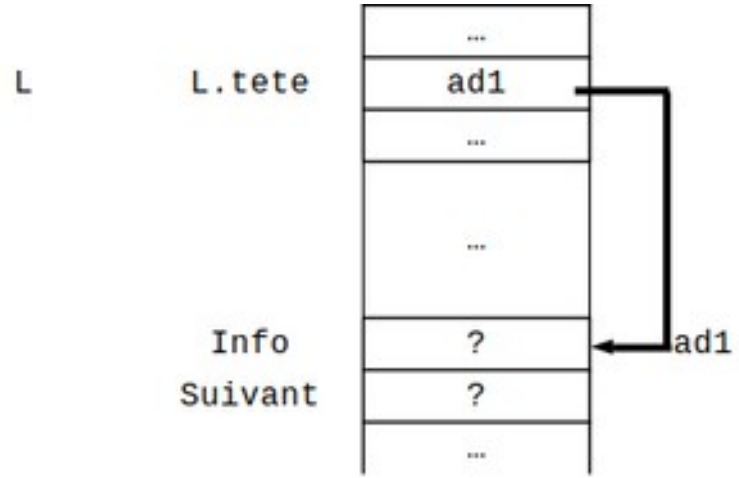
La liste ne contient aucune donnée, l'attribut `tete` de la liste `L` "ne pointe sur rien".

Une bonne pratique est de l'initialiser avec `NULL` :


```
L.tete = NULL;
```

Cela permet de savoir simplement si une liste est vide, il suffit de tester si sa tete vaut `NULL`.

Ensuite on ajoute une valeur dans la liste, pour cela on crée un premier maillon sur lequel pointe la tete de la liste :
`L.tete = new Maillon:`



`ad1` représente l'adresse du maillon alloué.

On va maintenant mettre la valeur 12 dans ce premier maillon :
il faut donc mettre cette valeur dans le champ Info du maillon pointé par `L.tete` :
pour cela on accède au maillon avec l'opérateur * qui permet d'accéder à l'objet pointé :

`*(L.tete)` est le maillon puisque `L.tete` est le pointeur sur le maillon

et comme on veut accéder au champ Info on l'ajoute en utilisant l'opérateur '.' :

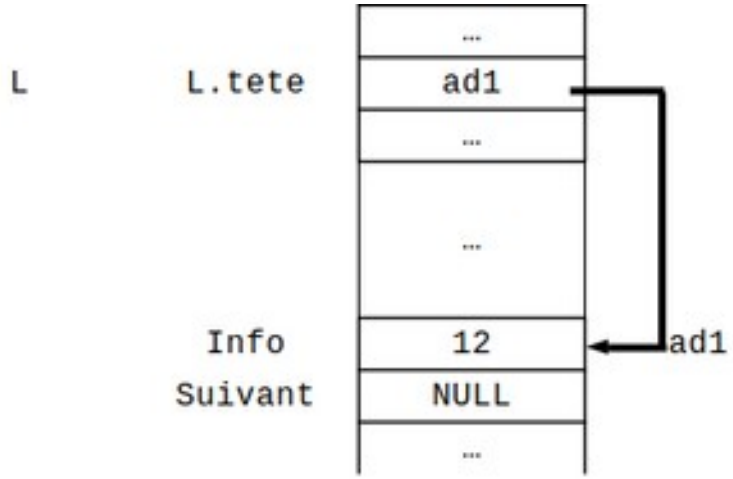
```
*(L.tete).Info = 12;
```

Remarque il existe un opérateur qui simplifie cette écriture, c'est l'opérateur -> il s'agit d'une flèche écrite avec les symboles - et > qui représente bien la flèche du pointeur telle que sur le dessin précédent.
L'expression devient :

```
L.tete->Info = 12;
```

Il faut aussi indiquer qu'il n'y a qu'un seul maillon, c'est à dire que le premier Maillon n'a pas de suivant, en mettant NULL dedans.

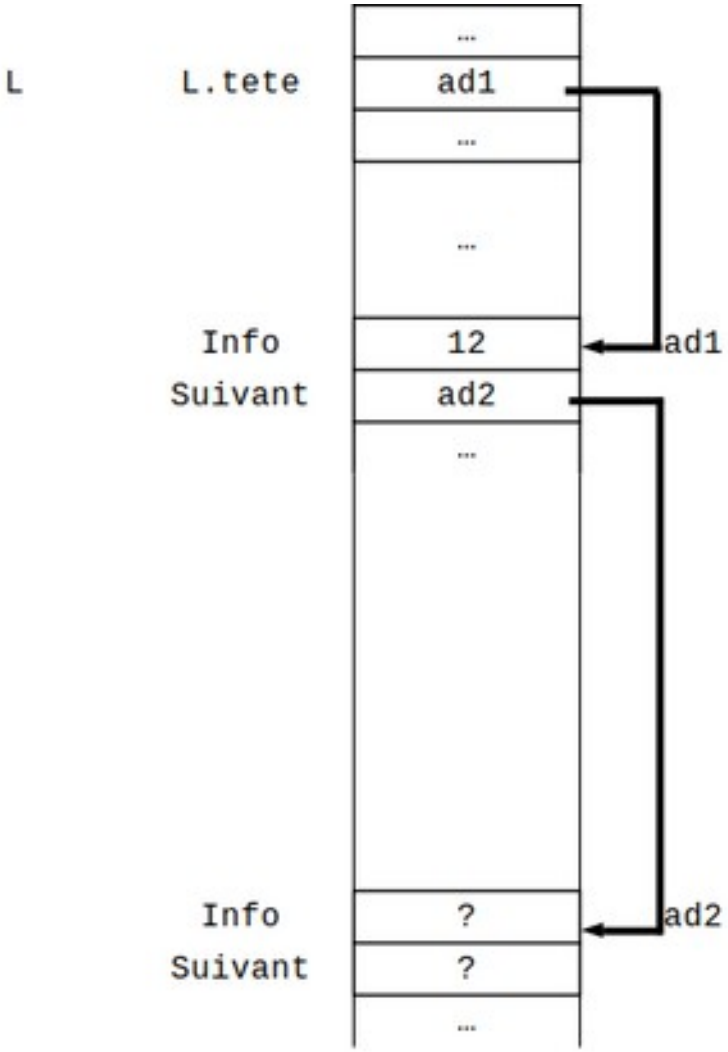
```
L.tete -> suivant = NULL;
```



Pour ajouter une deuxième valeur (par exemple 97) il faut faire le même genre de manipulation, mais attention, on n'a toujours accès qu'au pointeur `tete` par l'intermédiaire de la liste `L` :

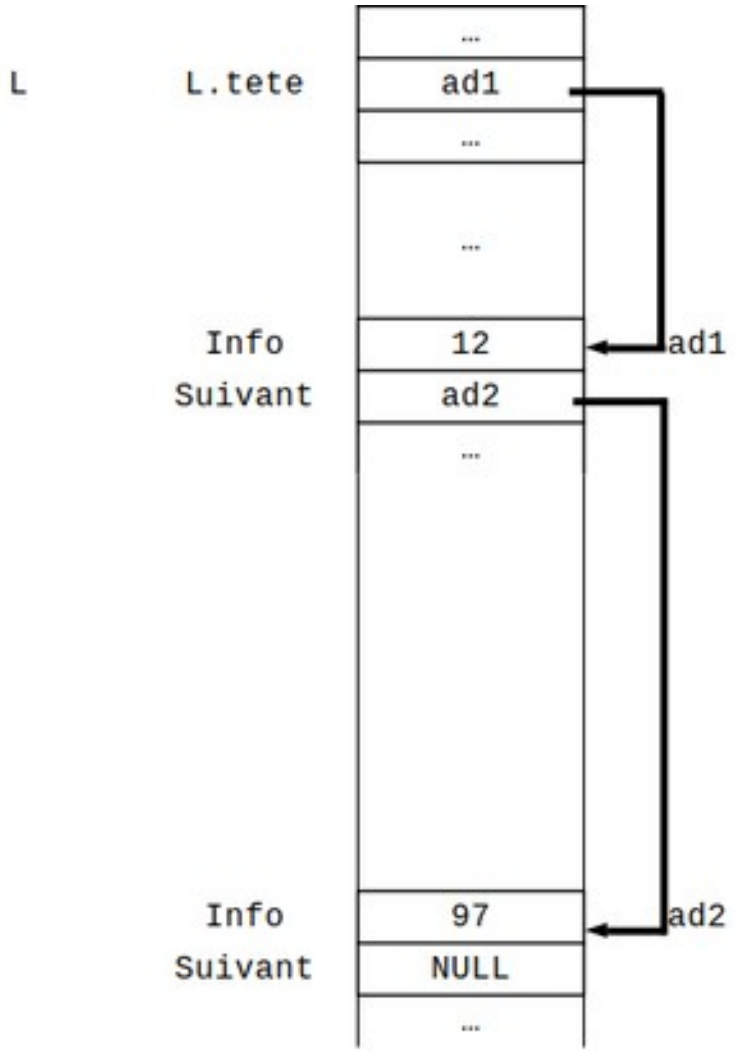
Création d'un maillon :

```
L.tete -> suivant = new Maillon;
```



Puis remplissage :

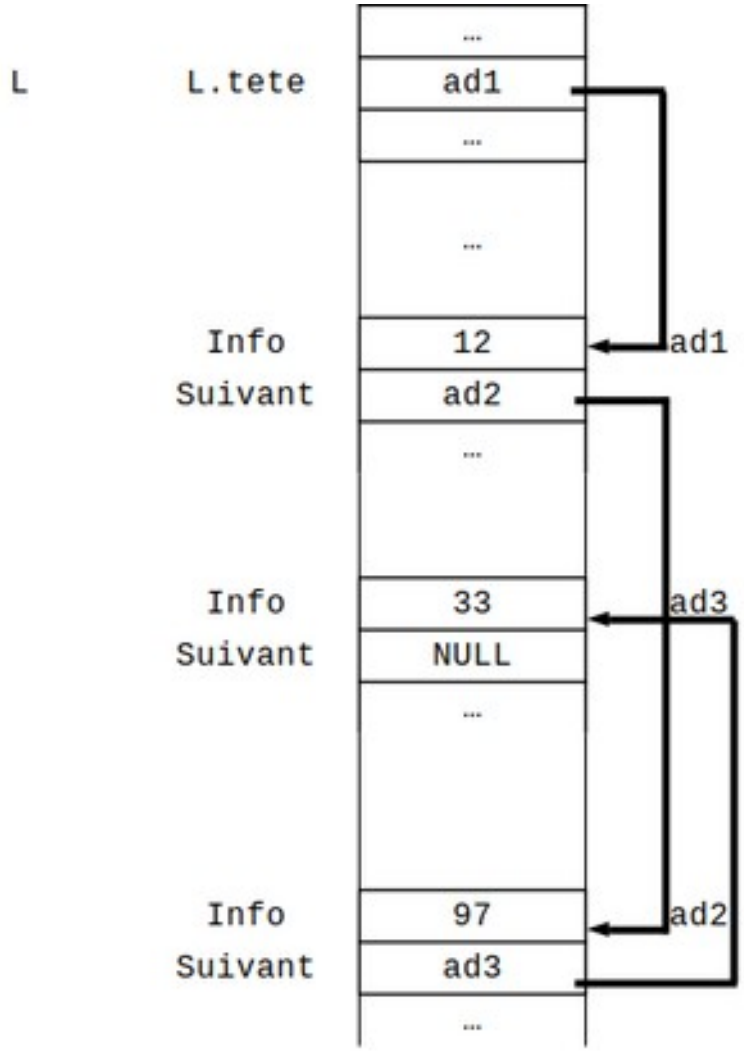
```
L.tete -> suivant -> Info = 97;
L.tete -> suivant -> suivant = NULL;
```



Un dernier :

```
L.tete -> suivant -> suivant = new Maillon;
L.tete -> suivant -> suivant -> Info = 33;
L.tete -> suivant -> suivant ->suivant = NULL;
```

Attention dans ce qui précède les accès aux données tete, suivant, Info ne sont pas toujours possibles, en effet ils sont "private" par défaut.
Pour que cela soit possible il faut
- mauvaise solutions : kles rendre "public"
- bonne solution : mettre ce code dans des fonctions des classe Maillon et Liste !



Programmation de toutes ces opérations :
Initialisation d'une liste : le bon endroit pour réaliser l'initialisation d'une liste est ... le constructeur de la liste.

```
Liste::Liste()
{
  tete = NULL;
}
```

On peut faire directement pointer la tête de la liste sur un maillon fourni au constructeur :

```
Liste::Liste(Maillon * M)
{
  tete = M;
}
```

Pour le maillon plusieurs solutions, on peut l'initialiser par des constructeurs, ou par des fonctions de la classe liste :

```
// Constructeur de Maillon sans valeur
Maillon::Maillon()
{
  suivant = NULL;
}

// Constructeur de Maillon avec valeur
Maillon::Maillon(int V)
{
  Info = V;
  suivant = NULL;
}

// Constructeur de Maillon avec valeur et suivant
Maillon::Maillon(int V, Maillon * S)
{
  Info = V;
  suivant = S;
}
```

Parcours de la liste et affichage de son contenu :

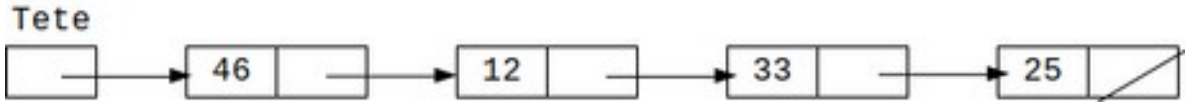
On va programmer une boucle qui va parcourir la liste maillon par maillon en commençant par le premier, c'est à dire par la tête de liste.
À chaque tour de boucle on affiche l'info pointée par le pointeur puis on fait pointer ce pointeur sur le maillon suivant.
Le traitement s'arrête quand il n'y a plus de maillon suivant.

Puisqu'on parle de la liste c'est une fonction de la classe Liste :

```
Liste :: Afficher()
{
// Déclaration d'un pointeur de Maillon
  Maillon * P;

// Initialisation de ce pointeur au debut de la liste
  P = tete;

// Mise en pace de la boucle
// on affiche tant qu'il y a des maillons ou tant que P pointe sur un maillon
// P pointe sur un maillon s'il n'est pas egal a NULL
  while (P != NULL)
  {
// Affichage de l'info du maillon pointe par P
    cout << P -> Info << "   ";

// Passage au maillon suivant de P
    P = P -> suivant;
  }
  cout << endl;
}
```

Ajout d'une valeur en début de liste, on ajoute 46 au début de cette liste :

Pour obtenir :



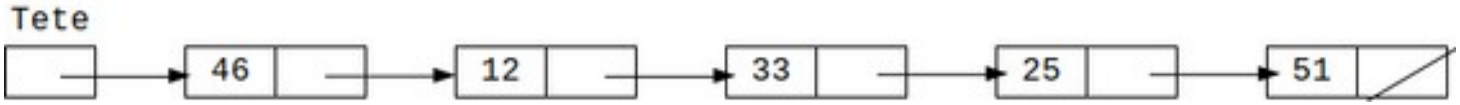
Attention il ne faut pas perdre la liste déjà présente

```
Liste::ajout_debut(int V)
{
  Maillon * P;
  P = new Maillon();
  P -> Info = V;
  P -> suivant = tete;
  tete = P;
}
```

Ce qui s'écrit simplement en une ligne :

```
Liste::ajout_debut(int V)
{
  tete = new Maillon(V,tete);
}
```

Ajout d'une valeur en fin de liste, on ajoute 51 à la fin :

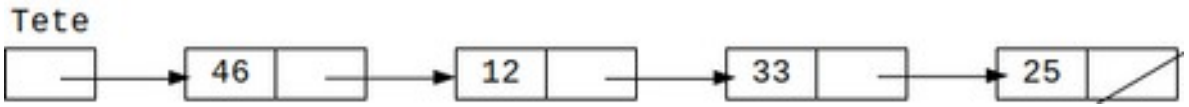


Il faut commencer par aller à la fin de la liste, il faut donc un pointeur qui part de tete et qui va jusqu'à la fin. Mais attention le nouveau maillon doit être pointé par le champ suivant du dernier Maillon, et pour cela il faut qu'on pointe dessus

pour pouvoir écrire

```
PP -> suivant = new Maillon(51);
```

Le plus simple est d'avoir 2 pointeurs qui se suivent quand le premier sera à NULL, le second sera sur le dernier Maillon.




```
Liste::ajout_fin(int V)
{
  Maillon * P, * PP; // PP pour pointeur precedent !
  P = tete;
  PP = NULL;
  while (P != NULL)
  {
    PP = P;
    P = P -> suivant;
  }
// Si la liste était vide, on est pas entré dans la boucle et PP est resté à NULL,
// alors il faut ajouter en tete

  if (PP = NULL)
    tete = new Maillon(V);
// sinon P est arrivé à NULL et PP pointe sur le dernier Maillon on peut donc ajout un
// Maillon à la fin

  else
    PP -> suivant = new Maillon(V);
}
```