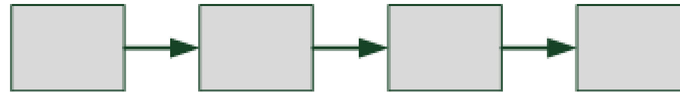
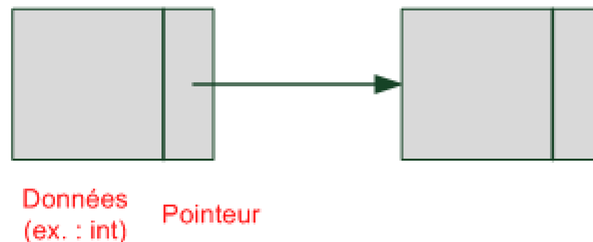
Les listes chaînées (suite)

Une liste chaînée est un moyen d'organiser une série de données en mémoire. Cela consiste à assembler des structures en les liants entre elles à l'aide de pointeurs. On pourrait les représenter comme ceci :



Chaque élément peut contenir ce que l'on veut : un ou plusieurs `int`, `double`… En plus de cela, chaque élément possède un pointeur vers l'élément suivant (fig. suivante).



Contrairement aux tableaux, les éléments d'une liste chaînée ne sont pas placés côte à côte dans la mémoire. Chaque case pointe vers une autre case en mémoire qui n'est pas nécessairement stockée juste à côté.

1 - Construction d'une liste chaînée

Un élément de la liste

Pour nos exemples, nous allons créer une liste chaînée de nombres entiers. Chaque élément de la liste aura la forme de la structure suivante :

```
typedef struct Element Element;
struct Element
{
    int nombre;
    Element *suivant;
};
```

Nous avons créé ici un élément d'une liste chaînée, que nous avons vue plus tôt. Que contient cette structure ?

- Une donnée, ici un nombre de type `int` : on pourrait remplacer cela par n'importe quelle autre donnée (un `double`, un tableau…). Cela correspond à ce que vous voulez stocker, c'est à vous de l'adapter en fonction des besoins de votre programme.

  Si on veut travailler de manière générique, l'idéal est de faire un pointeur sur void : `void*`. Cela permet de faire pointer vers n'importe quel type de données.

- Un pointeur vers un élément du même type appelé `suivant`. C'est ce qui permet de lier les éléments les uns aux autres : chaque élément « sait » où se trouve l'élément suivant en mémoire. Les cases ne sont pas côte à côte en mémoire. C'est la grande différence

par rapport aux tableaux. Cela offre davantage de souplesse car on peut plus facilement ajouter de nouvelles cases par la suite au besoin.

En revanche, il ne sait pas quel est l'élément précédent, il est donc impossible de revenir en arrière à partir d'un élément avec ce type de liste. On parle de liste « simplement chaînée », alors que les listes « doublement chaînées » ont des pointeurs dans les deux sens et n'ont pas ce défaut. Elles sont néanmoins plus complexes.

2 - La structure de contrôle

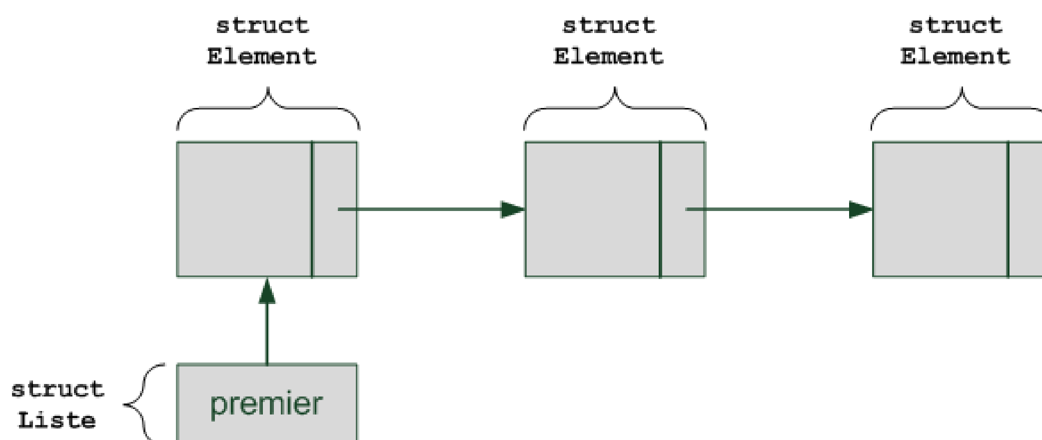
En plus de la structure qu'on vient de créer (que l'on dupliquera autant de fois qu'il y a d'éléments), nous allons avoir besoin d'une autre structure pour contrôler l'ensemble de la liste chaînée. Elle aura la forme suivante :

```
typedef struct Liste Liste;
struct Liste
{
    Element *premier;
};
```

Cette structure `Liste` contient un pointeur vers le premier élément de la liste. En effet, il faut conserver l'adresse du premier élément pour savoir où commence la liste. Si on connaît le premier élément, on peut retrouver tous les autres en « sautant » d'élément en élément à l'aide des pointeurs `suivant`.

Une structure composée d'une seule sous-variable n'est en général pas très utile. Néanmoins, on aura besoin d'y ajouter des sous-variables plus tard. On pourrait par exemple y stocker en plus la taille de la liste, c'est-à-dire le nombre d'éléments qu'elle contient.

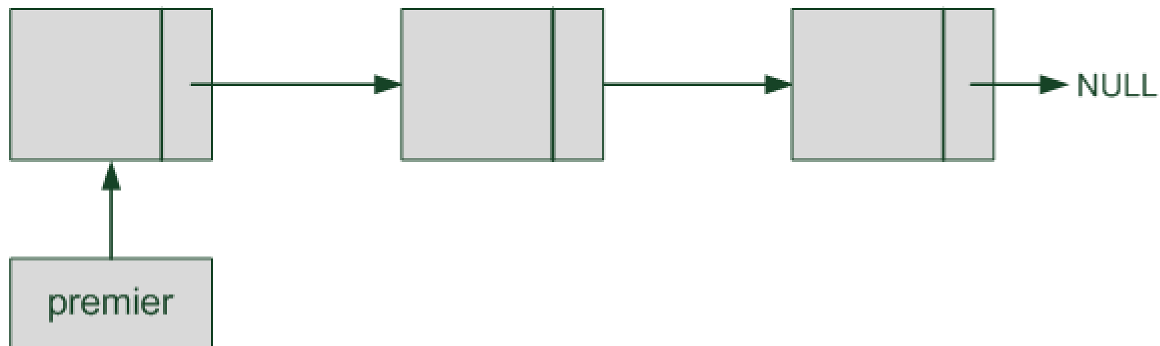
Nous n'aurons besoin de créer qu'un seul exemplaire de la structure `Liste`. Elle permet de contrôler toute la liste (fig. suivante).



3 - Le dernier élément de la liste

Notre schéma est presque complet. Il manque une dernière chose : on aimerait retenir le dernier élément de la liste. En effet, il faudra bien arrêter de parcourir la liste à un moment donné. Avec quoi pourrait-on signifier à notre programme « Stop, ceci est le dernier élément » ?

Il serait possible d'ajouter dans la structure `Liste` un pointeur vers le dernier `Element`. Toutefois, il y a encore plus simple : il suffit de faire pointer le dernier élément de la liste vers `NULL`, c'est-à-dire de mettre son pointeur `suivant` à `NULL`. Cela nous permet de réaliser un schéma enfin complet de notre structure de liste chaînée (fig. suivante).



4 - Les fonctions de gestion de la liste

Nous avons créé deux structures qui permettent de gérer une liste chaînée :

- `Element`, qui correspond à un élément de la liste et que l'on peut dupliquer autant de fois que nécessaire ;
- `Liste`, qui contrôle l'ensemble de la liste. Nous n'en aurons besoin qu'en un seul exemplaire.

C'est bien, mais il manque encore l'essentiel : les fonctions qui vont manipuler la liste chaînée. En effet, on ne va pas modifier « à la main » le contenu des structures à chaque fois qu'on en a besoin ! Il est plus sage et plus propre de passer par des fonctions qui automatisent le travail. Encore faut-il les créer.

On aura besoin de fonctions pour :

- initialiser la liste ;
- ajouter un élément ;
- supprimer un élément ;
- afficher le contenu de la liste ;
- supprimer la liste entière.

On pourrait créer d'autres fonctions (par exemple pour calculer la taille de la liste) mais elles sont moins indispensables. Nous allons ici nous concentrer sur celles citées plus haut, ce qui nous fera déjà une bonne base.

4.1 - Initialiser la liste

La fonction d'initialisation est la toute première que l'on doit appeler. Elle crée la structure de contrôle et le premier élément de la liste.

```
Liste *initialisation()
{
    Liste *liste = malloc(sizeof(*liste));
    Element *element = malloc(sizeof(*element));
```

```
    if (liste == NULL || element == NULL)
    {
        exit(EXIT_FAILURE);
    }

    element->nombre = 0;
    element->suivant = NULL;
    liste->premier = element;

    return liste;
}
```

On commence par créer la structure de contrôle `liste`.

Notez que le type de données est `Liste` et que la variable s'appelle `liste`. La majuscule permet de les différencier.

On alloue dynamiquement la structure de contrôle avec un `malloc`. La taille à allouer est calculée automatiquement avec `sizeof(*liste)`. L'ordinateur saura qu'il doit allouer l'espace nécessaire au stockage de la structure `Liste`.

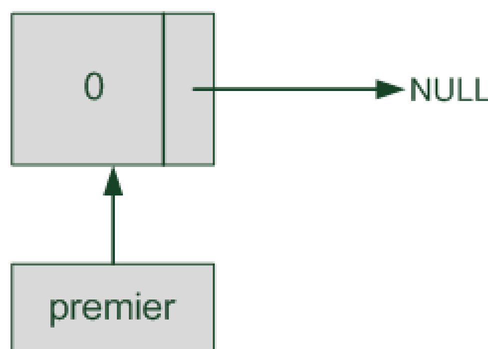
On aurait aussi pu écrire `sizeof(Liste)`, mais si plus tard on décide de modifier le type du pointeur `liste`, on devra aussi adapter le `sizeof`.

On alloue ensuite de la même manière la mémoire nécessaire au stockage du premier élément. On vérifie si les allocations dynamiques ont fonctionné. En cas d'erreur, on arrête immédiatement le programme en faisant appel à `exit()`.

Si tout s'est bien passé, on définit les valeurs de notre premier élément :

- la donnée `nombre` est mise à 0 par défaut ;
- le pointeur `suivant` pointe vers `NULL` car le premier élément de notre liste est aussi le dernier pour le moment. Comme on l'a vu plus tôt, le dernier élément doit pointer vers `NULL` pour signaler qu'il est en fin de liste.

Nous avons donc maintenant réussi à créer en mémoire une liste composée d'un seul élément et ayant une forme semblable à la fig. suivante.

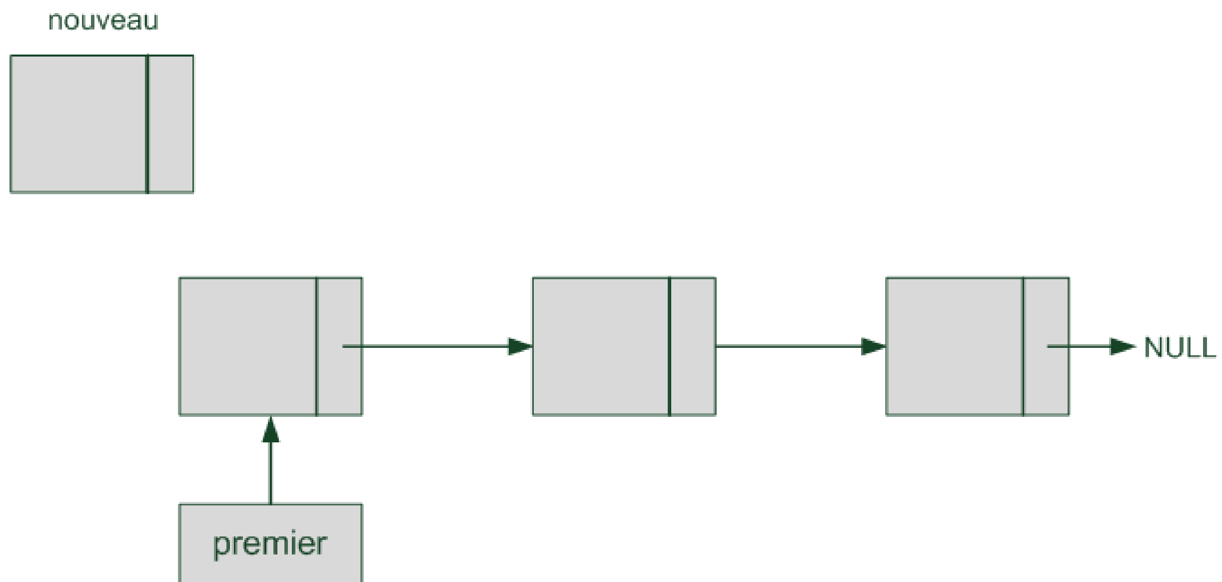


4.2 - Ajouter un élément

Ici, les choses se compliquent un peu. Où va-t-on ajouter un nouvel élément ? Au début de la liste, à la fin, au milieu ?

La réponse est qu'on a le choix. Libre à nous de décider ce que nous faisons. Pour cet exemple, on procèdera à l'ajout d'un élément en début de liste.

Nous devons créer une fonction capable d'insérer un nouvel élément en début de liste. Pour nous mettre en situation, imaginons un cas semblable à la fig. suivante : la liste est composée de trois éléments et on souhaite en ajouter un nouveau au début.



Il va falloir adapter le pointeur `premier` de la liste ainsi que le pointeur `suivant` de notre nouvel élément pour « insérer » correctement celui-ci dans la liste.

```
void insertion(Liste *liste, int nvNombre)
{
    /* Création du nouvel élément */
    Element *nouveau = malloc(sizeof(*nouveau));
    if (liste == NULL || nouveau == NULL)
    {
        exit(EXIT_FAILURE);
    }
    nouveau->nombre = nvNombre;

    /* Insertion de l'élément au début de la liste */
    nouveau->suivant = liste->premier;
    liste->premier = nouveau;
}
```

La fonction `insertion()` prend en paramètre l'élément de contrôle `liste` (qui contient l'adresse du premier élément) et le nombre à stocker dans le nouvel élément que l'on va créer.

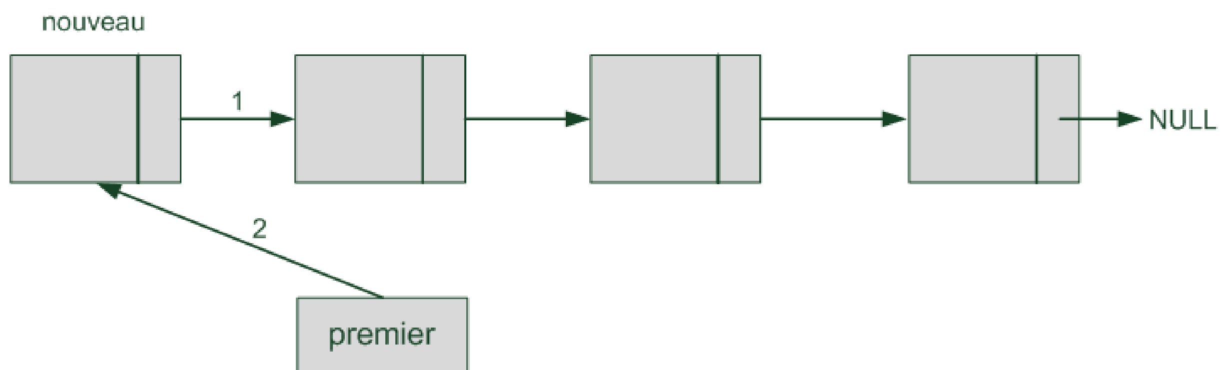
Dans un premier temps, on alloue l'espace nécessaire au stockage du nouvel élément et on y place le nouveau nombre `nvNombre`. Il reste alors une étape délicate : l'insertion du nouvel élément dans la liste chaînée.

Nous avons ici choisi pour simplifier d'insérer l'élément en début de liste. Pour mettre à jour correctement les pointeurs, nous devons procéder dans cet ordre précis :

1. faire pointer notre nouvel élément vers son futur successeur, qui est l'actuel premier élément de la liste ;
2. faire pointer le pointeur `premier` vers notre nouvel élément.

On ne peut pas suivre ces étapes dans l'ordre inverse ! En effet, si vous faites d'abord pointer `premier` vers notre nouvel élément, vous perdez l'adresse du premier élément de la liste ! Faites le test, vous comprendrez de suite pourquoi l'inverse est impossible.

Cela aura pour effet d'insérer correctement notre nouvel élément dans la liste chaînée (fig. suivante) !



4.3 - Supprimer un élément

De même que pour l'insertion, nous allons ici nous concentrer sur la suppression du premier élément de la liste. Il est techniquement possible de supprimer un élément précis au milieu de la liste.

La suppression ne pose pas de difficulté supplémentaire. Il faut cependant bien adapter les pointeurs de la liste dans le bon ordre pour ne « perdre » aucune information.

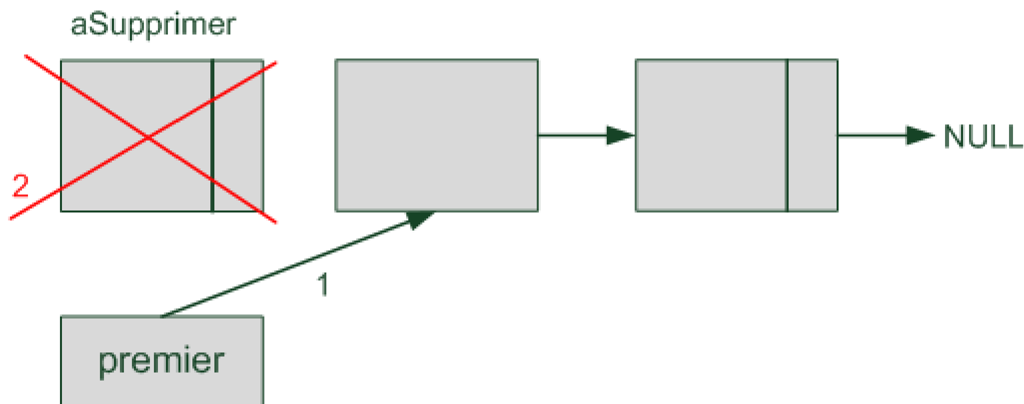
```
void suppression(Liste *liste)
{
    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }

    if (liste->premier != NULL)
    {
        Element *aSupprimer = liste->premier;
        liste->premier = liste->premier->suivant;
        free(aSupprimer);
    }
}
```

On commence par vérifier que le pointeur qu'on nous envoie n'est pas `NULL`, sinon on ne peut pas travailler. On vérifie ensuite qu'il y a au moins un élément dans la liste, sinon il n'y a rien à faire.

Ces vérifications effectuées, on peut sauvegarder l'adresse de l'élément à supprimer dans un pointeur `aSupprimer`. On adapte ensuite le pointeur `premier` vers le nouveau premier élément, qui est actuellement en seconde position de la liste chaînée.

Il ne reste plus qu'à supprimer l'élément correspondant à notre pointeur `aSupprimer` avec un `free` (fig. suivante).



Cette fonction est courte mais sauriez-vous la réécrire ? Il faut bien comprendre qu'on doit faire les choses dans un ordre précis :

1. faire pointer `premier` vers le second élément ;
2. supprimer le premier élément avec un `free`.

Si on faisait l'inverse, on perdrait l'adresse du second élément !

4.4 - Afficher la liste chaînée

Pour bien visualiser ce que contient notre liste chaînée, une fonction d'affichage serait idéale ! Il suffit de partir du premier élément et d'afficher chaque élément un à un en « sautant » de bloc en bloc.

```
void afficherListe(Liste *liste)
{
    if (liste == NULL)
    {
        exit(EXIT_FAILURE);
    }

    Element *actuel = liste->premier;

    while (actuel != NULL)
    {
        printf("%d -> ", actuel->nombre);
        actuel = actuel->suivant;
    }
    printf("NULL\n");
```

```
}
```

Cette fonction est simple : on part du premier élément et on affiche le contenu de chaque élément de la liste (un nombre). On se sert du pointeur `suivant` pour passer à l'élément qui suit à chaque fois.

On peut s'amuser à tester la création de notre liste chaînée et son affichage avec un `main` :

```
int main()
{
    Liste *maListe = initialisation();

    insertion(maListe, 4);
    insertion(maListe, 8);
    insertion(maListe, 15);
    suppression(maListe);

    afficherListe(maListe);

    return 0;
}
```

En plus du premier élément (que l'on a laissé ici à 0), on en ajoute trois nouveaux à cette liste. Puis on en supprime un. Au final, le contenu de la liste chaînée sera donc :

```
8 -> 4 -> 0 -> NULL
```

Travaux Pratiques (TP)

Compiler les différentes portions de programme contenues dans ce document pour en faire un programme de création et de gestion de listes simplement chainées. Utilise une structure de choix pour permettre de choisir les actions à effectuer sur les listes.

Source *: www.openclassrooms.com*