Les listes

Une liste est une structure permettant de stocker une collection de données de même type. Contrairement aux tableaux, l'espace mémoire utilisé par une liste n'est pas contigu. La taille d'une liste est inconnue à priori ; elle peut contenir autant d'éléments que l'on veut. On ne peut accéder directement qu'au premier élément d'une liste. Pour accéder au i^{ème} d'une liste, il faut parcourir tous les éléments de la liste jusqu'au i^{ème}.

Les listes servent à gérer un ensemble de données, un peu comme les tableaux. Elles sont cependant plus efficaces pour réaliser des opérations comme l'insertion et la suppression d'éléments.
Elles utilisent par ailleurs l'allocation dynamique de mémoire et peuvent avoir une taille qui varie pendant l'exécution. L'allocation (ou la libération de mémoire) se fait élément par élément.
Les opérations sur une liste peuvent être :

- Créer une liste ;
- Supprimer une liste ;
- Rechercher un élément particulier ;
- Insérer un élément (en début, en fin ou au milieu) ;
- Supprimer un élément particulier ;
- Permuter deux éléments ;
- Concaténer deux listes ;
- ...

Les listes peuvent par ailleurs être :

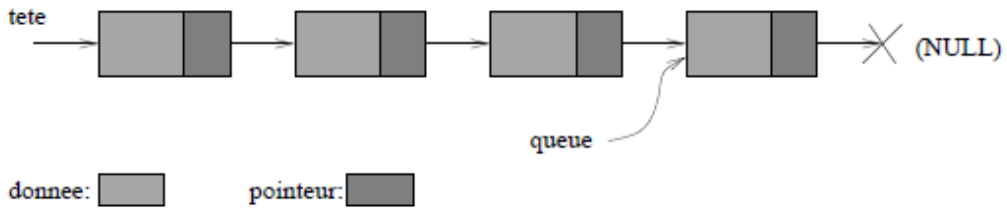
- simplement chaînées ;
- doublement chaînées ;
- circulaires (chainage simple ou double).

1.1 - Listes simplement chaînées

Une liste simplement chainée est composée d'éléments distincts liés par un simple pointeur.
Chaque élément d'une liste simplement chainée est formé de deux parties :

- un champ contenant la donnée (ou un pointeur vers celle-ci) ;
- un pointeur vers l'élément suivant de la liste.

Le premier élément d'une liste est sa tête, le dernier sa queue. Le pointeur du dernier élément est initialisé une valeur sentinelle, par exemple la valeur NULL en C.



Pour accéder à un élément d'une liste simplement chainée, on part de la tête et on passe d'un élément à l'autre à l'aide du pointeur suivant associe à chaque élément.
En pratique, les éléments étant créés par allocation dynamique, ne sont pas contigus en mémoire contrairement à un tableau. La suppression d'un élément sans précaution ne permet plus d'accéder aux éléments suivants.
D'autre part, une liste simplement chainée ne peut être parcourue que dans un sens (de la tête vers la queue).

Exemple d'implémentation sous forme d'une structure en C:

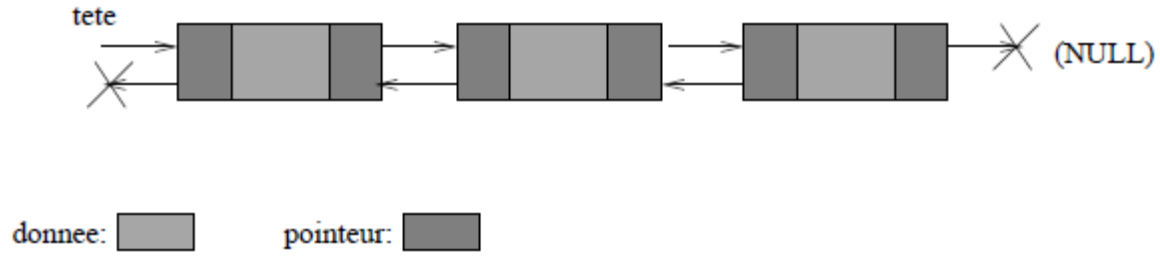
```
struct s_element
{
int donnee;
struct s_element* suivant;
};
typedef struct s_element t_element;
```

1.2 - Listes doublement chaînées

Les listes doublement chaînées sont constituées d'éléments comportant trois champs:

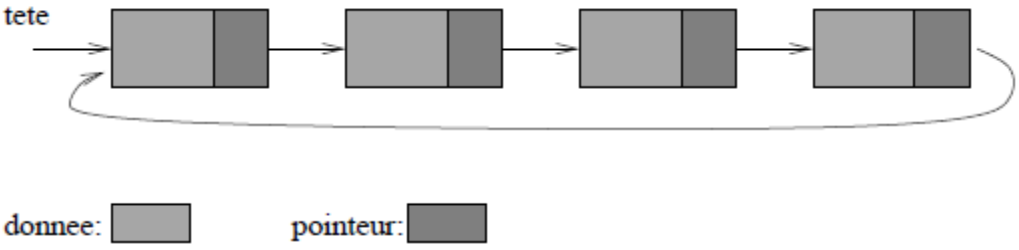
- un champ contenant la donnée (ou un pointeur vers celle-ci) ;
- un pointeur vers l'élément suivant de la liste ;
- un pointeur vers l'élément précèdent de la liste.

Elles peuvent donc être parcourues dans les deux sens.



1.3 - Listes circulaires

Une liste circulaire peut être simplement ou doublement chainée. Sa particularité est de ne pas comporter de queue. Le dernier élément de la liste pointe vers le premier. Un élément possède donc toujours un suivant.



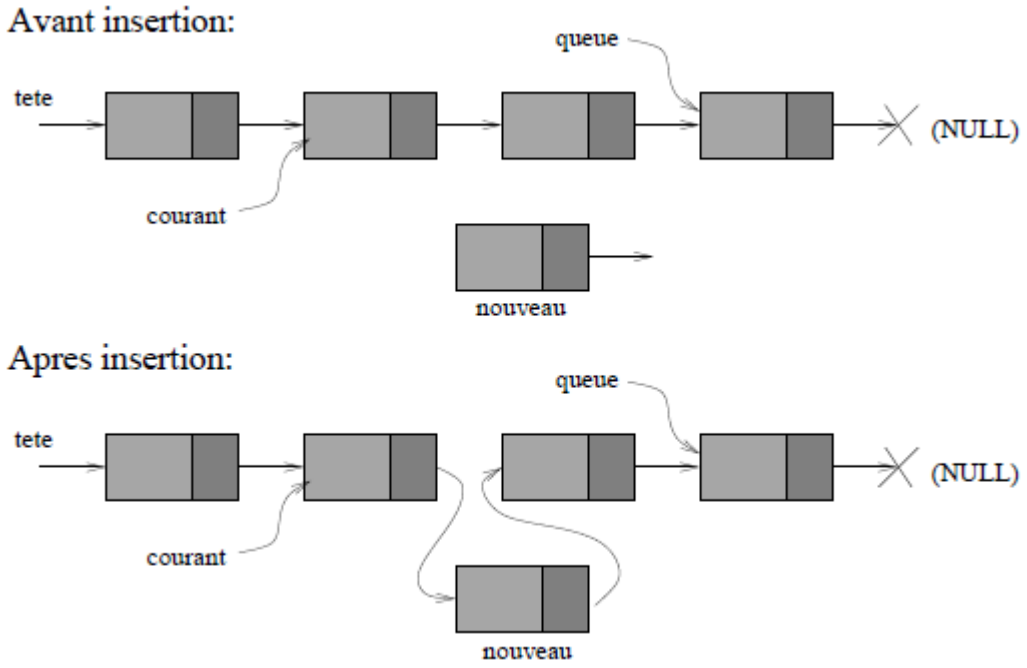
1.4 - Operations sur une liste (cas des listes simplement chainées)

1.4.1 - Insertion d'un élément

L'insertion d'un élément dans une liste peut se faire :

- en tête de liste
- en queue de liste
- n'importe où (à une position fixée par un pointeur dit « courant »)

L'exemple choisi est celui de l'insertion n'importe où (après l'élément référencé par le pointeur courant) :



Les opérations à effectuer sont :

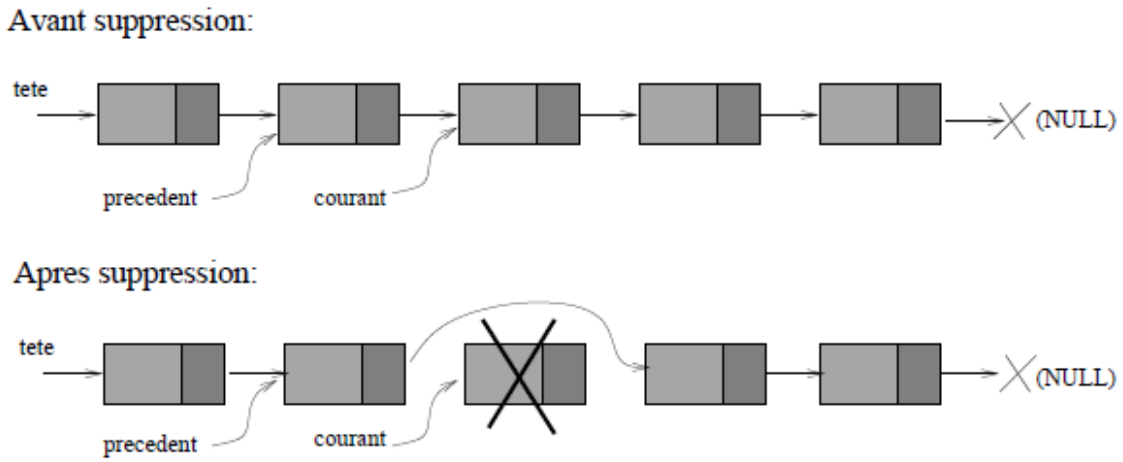
- allouer de la mémoire pour le nouvel élément
- copier les données
- faire pointer le nouvel élément vers l'élément suivant de celui pointé par courant (vers NULL s'il n'y a pas de suivant)
- faire pointer l'élément pointé par courant vers le nouvel élément

Le cas où la liste est vide (tète égale à NULL) doit être traité à part.

Exemple d'implémentation de la fonction d'insertion en C :

```
void insertion(t_element** tete, t_element* courant, int data)
{
t_element* nouveau;
nouveau=(t_element*) malloc(sizeof(t_element));
nouveau->donnee=data;
if (courant!=NULL)
{
nouveau->suivant=courant->suivant;
courant->suivant=nouveau;
}
else /* insertion en tete ou liste vide */
{
nouveau->suivant=*tete;
*tete=nouveau;
}
}
```

1.4.2 - Suppression d'un élément



Dans le cas d'une liste simplement chainée, la fonction de suppression demande un peu de réflexion pour être implémentée.
Par exemple, une fonction permettant de supprimer l'élément pointé par courant pourrait avoir cette interface:
void suppression(t_element** tete, t_element* courant)
Le passage du pointeur de tête par adresse est nécessaire pour pouvoir le modifier si la suppression rend la liste vide ou si l'élément à supprimer était le premier de la liste.
On doit d'abord distinguer s'il s'agit du 1er élément de la liste ou pas:
if (courant!=*tete) /* pas le premier element */

Dans ce cas il faut chercher le prédécesseur de courant. La liste étant simplement chainée, on n'a pas d'autre solution que de faire un parcours depuis la tête jusqu'à trouver le précédent de courant.

```
precedent=*tete;
while (precedent->suivant!=courant)
precedent=precedent->suivant;
```

On peut alors récupérer le lien contenu dans le champ suivant de l'élément à supprimer:

```
precedent->suivant=courant->suivant;
```

Dans le cas où c'est le premier élément que l'on supprime, on modifie simplement le pointeur de tête:

```
else /* suppression du 1er element */
*tete=courant->suivant;
```

Enfin, dans tous les cas on libère la mémoire:

```
free(courant);
```