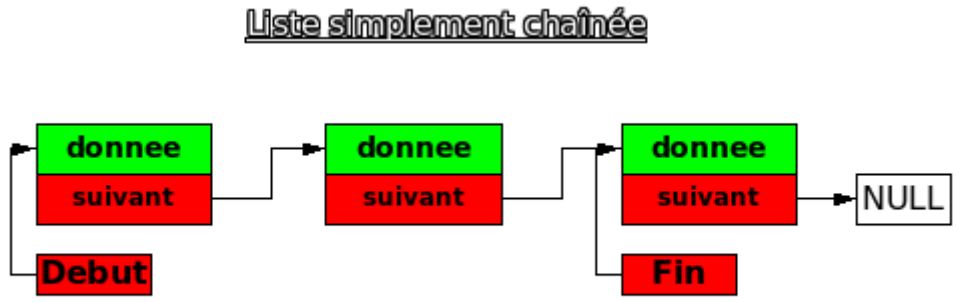
Chapitre 1 : Listes chaînées

1. Définition

Les listes chaînées sont des structures de données semblables aux tableaux sauf que l'accès à un élément ne se fait pas par *index* mais à l'aide d'un *pointeur*. L'allocation de la mémoire est faite au moment de l'exécution.

Dans une liste les éléments sont contigus en ce qui concerne l'enchaînement.



En revanche, par rapport aux tableaux où les éléments sont contigus dans la mémoire, les éléments d'une liste sont éparpillés dans la mémoire. La liaison entre les éléments se fait grâce à un pointeur. En réalité, dans la mémoire la représentation est aléatoire en fonction de l'espace alloué.

Le pointeur suivant du dernier élément doit pointer vers NULL (la fin de la liste).

Pour accéder à un élément la liste est parcourue en commençant avec la tête, le pointeur suivant permettant le déplacement vers le prochain élément. Le déplacement se fait dans une seule direction, du premier vers le dernier élément.
Si vous voulez vous déplacer dans les deux directions (en avant/en arrière) utilisez les listes doublement chaînées.

2. La construction du prototype d'un élément de la liste

Pour définir un élément de la liste le type *struct* sera utilisé. L'élément de la liste contiendra un champ data (pouvant être une structure si nécessaire) et un pointeur suivant. Le pointeur suivant doit être du même type que l'élément, sinon il ne pourra pas pointer vers l'élément.

Le pointeur "suivant" permettra l'accès vers le prochain élément.

```
Type structure data
      Type1 champ1
      Type2 champ2
      …
fin

Type structure element
      Data d
      element modifiable suivant
fin
```

Pour avoir le contrôle de la liste il est préférable de sauvegarder certaines informations: l'adresse du premier élément et du dernier élément ainsi que le nombre d'éléments.

Pour réaliser cela une autre structure sera utilisée (ce n'est pas obligatoire, des variables peuvent être utilisées).

```
Type structure liste
            Element modifiable tete;
            Element modifiable queue;
            entier taille;
}
```

Le pointeur `tete` contiendra l'adresse du premier élément de la liste.
Le pointeur `queue` contiendra l'adresse du dernier élément de la liste.
La variable `taille` contient le nombre d'éléments.

Quel que soit la position dans la liste, les pointeurs `tete` et `queue` pointent toujours respectivement vers le 1er et le dernier élément.
Le champ `taille` contiendra le nombre d'éléments de la liste quel que soit l'opération effectuée sur la liste.

3. Opérations sur les listes chaînées

A. Initialisation

Prototype de la fonction :
```
Vide fonction init (liste modifiable L);
```

Cette opération doit être faite avant toute autre opération sur la liste.

Elle initialise le pointeur `tete` et le pointeur `queue` avec le pointeur NULL, et la `taille` avec la valeur 0.

Le pointeur `NULL` est une adresse fictive ne correspondant à aucune zone de mémoire accessible. Dans la plupart des systèmes, il vaut 0.

La fonction
```
vide fonction init (liste modifiable L);

            L->tete <- NULL;
            L->queue <-  NULL;
            L->taille <- 0;
fin
```

B. Affichage de la liste

Pour afficher la liste entière il faut se positionner au début de la liste (le pointeur `tete` le permettra).
Ensuite en utilisant le pointeur suivant de chaque élément, la liste est parcourue du 1er vers le dernier élément.
La condition d'arrêt est donnée par le pointeur suivant du dernier élément qui vaut NULL.

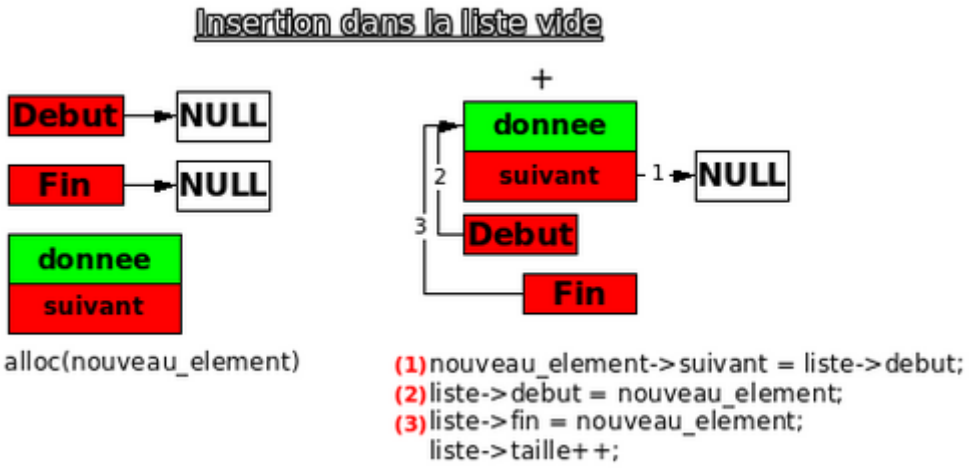
La fonction
```
vide fonction afficher(liste modifiable L)

      element modifiable courant;

      courant = L->tete;
      tantque (courant != liste->queue) faire

            Afficher(courant->data);
            courant <- courant->next;
      fin
fin
```

C. Insertion d'un élément dans la liste

1. Insertion dans une liste vide

Étapes :

- allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- le pointeur suivant du nouvel élément pointera vers NULL (vu que l'insertion est faite dans une liste vide on utilise l'adresse du pointeur debut qui vaut NULL)
- les pointeurs debut et fin pointeront vers le nouvel élément
- la taille est mise à jour

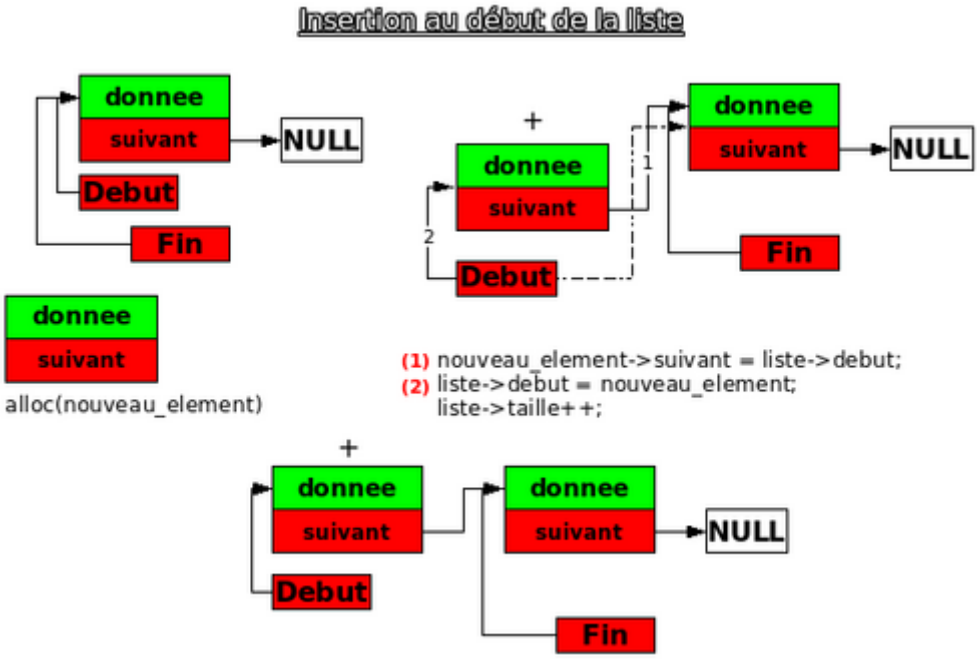



```
void creer(liste modifiable L, data D)

  element modifiable nouveau;
  allouer(nouveau)

  si (nouveau != NULL)

        nouveau->d <- D ;
        nouveau->next <- NULL;

        L->tete <- nouveau;
        L->queue <- nouveau;
        L->taille <- L->taille + 1;

  sinon

        Ecrire 'memoire insuffisante'
        sortir
  finsi
fin
```

2. Insertion en tête de liste

Étapes:

- allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- le pointeur suivant du nouvel élément pointe vers le 1er élément
- le pointeur debut pointe vers le nouvel élément
- le pointeur fin ne change pas
- la taille est incrémentée

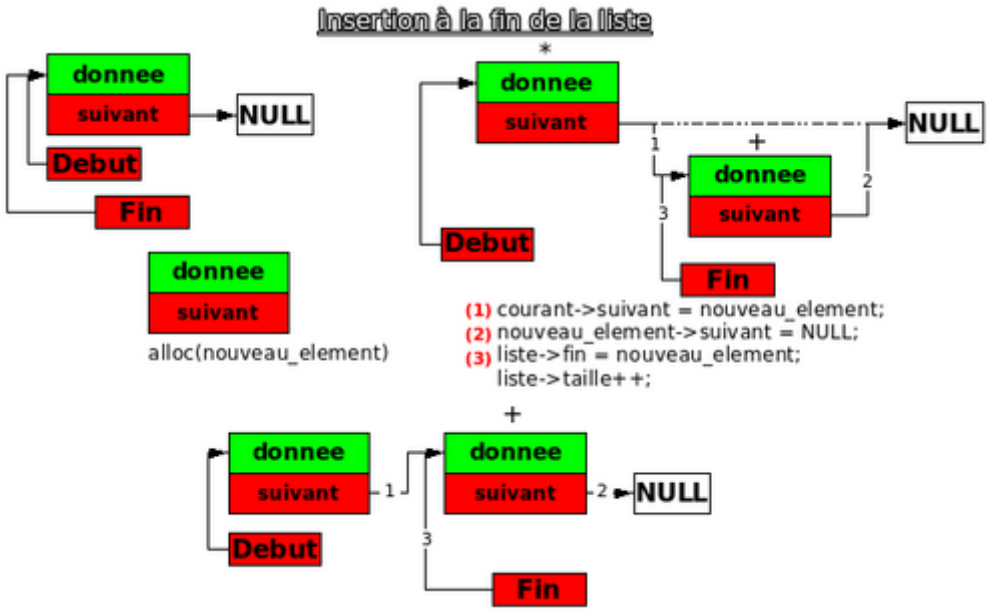



```
void inserer_en_tete(liste modifiable L, data D)

  element modifiable nouveau, courant
  allouer(nouveau)

  si (nouveau != NULL)

        courant <- liste->tete
        nouveau->d <- D
        nouveau->suivant <- courant

        L->tete <- nouveau
        L->taille <- L->taille + 1
  sinon
       .
       .
       .
  finsi
fin
```

3. Insertion en queue

Étapes:

- allocation de la mémoire pour le nouvel élément
- remplir le champ de données du nouvel élément
- le pointeur suivant du dernier élément pointe vers le nouvel élément
- le pointeur fin pointe vers le nouvel élément
- le pointeur debut ne change pas
- la taille est incrémentée




```
void inserer_en_queue(liste modifiable L, data D)

  element  modifiable nouveau, courant
  allouer(nouveau)

  si (nouveau != NULL)

        courant <- L->queue ;
        nouveau->d <- D ;
        courant->suivant <- nouveau ;
        nouveau->suivant <- NULL ;

        L->queue <- nouveau;
        L->taille <- L->taille + 1;
  sinon
        .
        .
        .
  finsi
fin
```