Cours « Complément de langage C pour l'Electronique »

III. Listes Chainées

1. Généralités

Une liste est un ensemble fini d'éléments notée L = e_1; e_2;…; e_n où e_1 est le premier élément, e_2 le deuxième, etc... Lorsque n=0 on dit que la liste est vide.

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 (remarque : Un tableau peut aussi être défini dynamiquement mais pour modifier sa taille, il faut en créer un nouveau, transférer les données puis supprimer l'ancien.).

L'allocation (ou la libération) 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 _n ou au milieu)
- Supprimer un élément particulier
- Permuter deux éléments
- Concaténer deux listes
- ...

Les listes peuvent par ailleurs être:
- simplement chaînées,
- doublement chaînées,
- circulaires (chaînage simple ou double).

2. Listes simplement chaînées

Une liste simplement chaînée est composée d'éléments distincts liés par un simple pointeur.

Chaque élément d'une liste simplement chaînée est formé de deux parties:
- un champ (ou plusieurs champs) 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 à une valeur sentinelle, par exemple la valeur NULL en C.

Pour accéder à un élément d'une liste simplement chaînée, on part de la tête et on passe d'un élément à l'autre à l'aide du pointeur suivant associé à chaque élément.

En pratique, les éléments étant crées 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 chaînée 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;
```

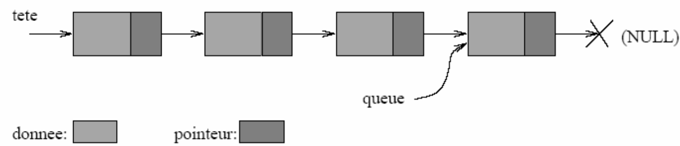

FIG. 3.1 – *Liste simplement chaînée*

3. 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 précédent de la liste.

Elles peuvent donc être parcourues dans les deux sens.

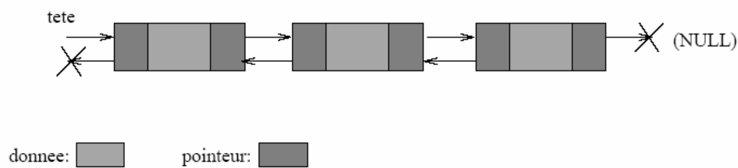

FIG. 3.2 – *Liste doublement chaînée*

4. Listes circulaires

Une liste circulaire peut être simplement ou doublement chaînée. 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.

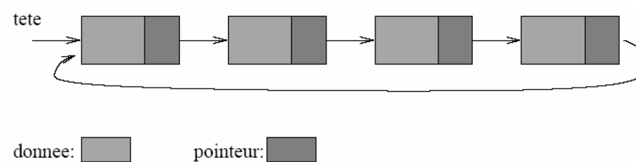

FIG. 3.3 – *Liste circulaire simplement chaînée*

5. Manipulation de liste

On ne décrira dans ce paragraphe que quelques opérations sur les listes simplement chaînées.

5.1 Insertion d'un élément

L'insertion d'un élément dans une liste peut se faire:
- en tête de liste (chaînage avant)
- en queue de liste (chaînage arrière)
- n'importe ou (à une position fixée par un pointeur dit courant)

Exemple d'implémentation :

Les opérations à effectuer sont (dans l'ordre!):
- allouer de la mémoire pour le nouvel élément (une fonction !)
- copier les données (une autre fonction !)
- insérer l'élément dans la liste (une troisième fonction !)

* Chaînage avant :

- premier exemple (renvoie de la nouvelle adresse de tête)

```
t_element *chainage_avant(t_element *tete, t_element *nouveau)
{
      if (tete==NULL) /*cas où la liste est vide*/
            return nouveau ;
      if (nouveau==NULL) /* cas où la nouveau est vide */
            return tete ;
      else /* cas normal */
      {
            nouveau->suivant=tete ;
            return nouveau ;
      }
}
```

- second exemple (passage par adresse de l'adresse de tête)

```
void chainage_avant(t_element **tete, t_element *nouveau)
{
      if (*tete==NULL)
            *tete=nouveau ;
      if (nouveau != NULL)
      {
            nouveau->suivant=*tete ;
            *tete= nouveau ;
      }
}
```

* Chaînage arrière :

```
t_element *chainage_arriere(t_element *tete, t_element *nouveau)
{
      t_element *courant = tete ; /* pointeur courant */

      if (tete==NULL) /* cas où la liste est vide */
            return nouveau ;

      /* détermination du dernier élément de la liste */
      while(courant->suivant != NULL) courant=courant->suivant ;

      /* ajout du nouveau */
      courant->suivant=nouveau ;
      return tete ;

}
```

* Insertion n'importe où :

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

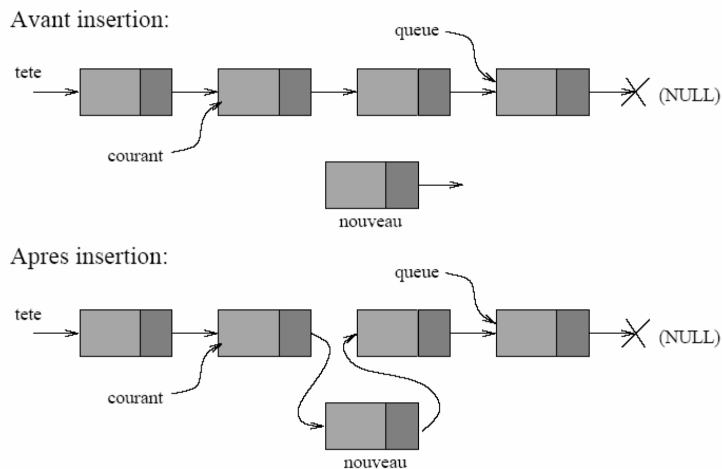

FIG. 3.4 – *Insertion*

Les opérations à effectuer sont (dans l'ordre!):
- 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 (tete égal à NULL) doit être traité à part.

```
t_element *insertion_nimporte_ou(t_element *tete, t_element *courant,
t_element *nouveau)
{
      if (nouveau==NULL) return tete ; /* gestion des erreurs */

      /* ajout du nouveau */
      if (courant !=NULL && tete !=NULL)
      {
            nouveau->suivant=courant->suivant ;
            courant->suivant=nouveau ;
            return tete ;
      }
      else /* cas d'une liste vide ou d'une insertion en tete */
      {
            nouveau->suivant=tete ;
            return nouveau ;
      }
}
```

5.2 Suppression d'un élément

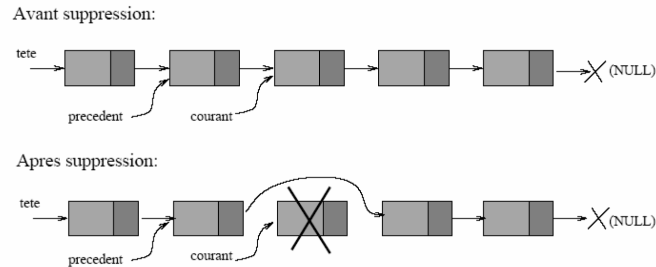

FIG. 3.5 – *Suppression*

Dans le cas d'une liste simplement chaînée, 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 :

```
t_element* suppression(t_element* tete, t_element* courant)
```

Le passage du pointeur de tête en argument de sortie 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 chaînée, 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);
```

5.3 Exemple d'utilisation (fonction `main()`)

Ne pas oublier les fonctions `alloue_element()`,`saisie_element()`, `recherche_element()`,…